1.5 UNION-FIND

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
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 not.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.
1.5 **Union-Find**

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Dynamic connectivity problem

Given a set of N objects, support two operation:

- Connect two objects.
- Is there a path connecting the two objects?

connect 4 and 3
connect 3 and 8
connect 6 and 5
connect 9 and 4
connect 2 and 1

are 0 and 7 connected?  ❌
are 8 and 9 connected?  ✔️
connect 5 and 0
connect 7 and 2
connect 6 and 1
connect 1 and 0
are 0 and 7 connected?  ✔️
A larger connectivity example

Q. Is there a path connecting $p$ and $q$?

A. Yes.
Applications involve manipulating objects of all types.

- Pixels in a digital photo.
- Computers in a network.
- Friends in a social network.
- Transistors in a computer chip.
- Elements in a mathematical set.
- Variable names in a Fortran program.
- 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

---

can use symbol table to translate from site names to integers: stay tuned (Chapter 3)
Modeling the connections

We assume "is connected to" is an equivalence relation:

- Reflexive: $p$ is connected to $p$.
- Symmetric: if $p$ is connected to $q$, then $q$ is connected to $p$.
- Transitive: if $p$ is connected to $q$ and $q$ is connected to $r$, then $p$ is connected to $r$.

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

\[
\begin{align*}
\{0\} & \quad \{1\} & \quad \{4, 5\} & \quad \{2, 3, 6, 7\} \\
0 & \quad 1 & \quad 4 & \quad 5 \\
\{4\} & \quad \{5\} & \quad \{2\} & \quad \{3\} \\
\{6\} & \quad \{7\} & \quad \{0\} & \quad \{1\} & \quad \{2\} & \quad \{3\} & \quad \{4\} & \quad \{5\} & \quad \{6\} & \quad \{7\} \\
3 \text{ connected components}
\end{align*}
\]
Implementing the operations

**Find.** In which component is object \( p \) ?

**Connected.** Are objects \( p \) and \( q \) in the same component?

**Union.** Replace components containing objects \( p \) and \( q \) with their union.

---

**Example:**

Before union:

```
{ 0 }  { 1 4 5 }  { 2 3 6 7 }
```

After union(2, 5):

```
{ 0 }  { 1 2 3 4 5 6 7 }
```

Components:

- Before: 3 connected components
- After: 2 connected components

---

**Graph representations:**

Before union:

```
0 -- 4
  |   |
  |   v
  1 -- 5
    |   |
    |   v
    2 -- 3
    |
    v
    6 -- 7
```

After union(2, 5):

```
0 -- 4
  |   |
  |   v
  1 -- 5
    |   |
    |   v
    2 -- 3
    |
    v
    6 -- 7
```
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.
- Union and find operations may be intermixed.

```java
public class UF {
    UF(int N) {
        // initialize union-find data structure with N singleton objects (0 to N – 1)
    }
    void union(int p, int q) {
        // add connection between p and q
    }
    int find(int p) {
        // component identifier for p (0 to N – 1)
    }
    boolean connected(int p, int q) {
        // are p and q in the same component?
    }
}
```

1-line implementation of `connected()`:
```java
public boolean connected(int p, int q) {
    return find(p) == find(q);
}
```
Dynamic-connectivity client

- Read in number of objects $N$ from standard input.
- Repeat:
  - read in pair of integers from standard input
  - if they are not yet connected, connect them and print out pair

```java
public static void main(String[] args)
{
    int N = StdIn.readInt();
    UF uf = new UF(N);
    while (!StdIn.isEmpty())
    {
        int p = StdIn.readInt();
        int q = StdIn.readInt();
        if (!uf.connected(p, q))
        {
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
    }
}
```

% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7

already connected
1.5 UNION-FIND

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-find  [eager approach]

Data structure.

- Integer array `id[]` of length \( N \).
- Interpretation: \( id[p] \) is the id of the component containing \( p \).

```plaintext
id[]: 0 1 1 8 8 0 0 1 8 8
```

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected
Quick-find  [eager approach]

Data structure.
- Integer array id[] of length N.
- Interpretation: id[p] is the id of the component containing p.

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

Find. What is the id of p?
Connected. Do p and q have the same id?

Union. To merge components containing p and q, change all entries whose id equals id[p] to id[q].

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

After union of 6 and 1

problem: many values can change
id[6] = 0; id[1] = 1
6 and 1 are not connected
Quick-find demo
Quick-find demo

id[]

<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>id[]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
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 object to itself (N array accesses)
- return the id of p (1 array access)
- change all entries with id[p] to id[q] (at most 2N + 2 array accesses)
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>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

order of growth of number of array accesses

Union is too expensive. It takes $N^2$ array accesses to process a sequence of $N$ union operations 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!

Quadratic algorithms don't scale with technology.

- New computer may be 10x as fast.
- But, has 10x as much memory ⇒
  want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!
1.5 UNION-FIND

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-union  [lazy approach]

Data structure.

- Integer array \( id[] \) of length \( N \).
- Interpretation: \( id[i] \) is parent of \( i \).
- Root of \( i \) is \( id[id[...id[i]...]] \).

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{id[]} & 0 & 1 & 9 & 4 & 9 & 6 & 6 & 7 & 8 & 9 \\
\end{array}
\]
Quick-union  [lazy approach]

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

Find. What is the root of $p$?

Connected. Do $p$ and $q$ have the same root?

Union. To merge components containing $p$ and $q$, set the id of $p$'s root to the id of $q$'s root.

```
<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>id[]</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>
</tr>
</tbody>
</table>
```

```
0 1 9 4 9 6 6 7 8 9
```

```
root of 3 is 9
root of 5 is 6
3 and 5 are not connected
```
Quick-union demo
Quick-union demo
public class QuickUnionUF
{
    private int[] id;

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

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

    public void union(int p, int q)
    {
        int i = find(p);
        int j = find(q);
        id[i] = j;
    }
}
Quick-union is also 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>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>N †</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

† includes cost of finding roots

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/connected too expensive (could be \(N\) array accesses).
1.5 Union-Find

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Improvement 1: weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

![Diagram showing the process of weighted quick-union comparison between two trees with labels and annotations.](image-url)
Weighted quick-union demo

id[]

0 1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9
Weighted quick-union demo
Quick-union and weighted quick-union example

Quick-union and weighted quick-union (100 sites, 88 union() operations)

average distance to root: 5.11

average distance to root: 1.52

Quick-union and weighted quick-union (100 sites, 88 union() operations)
Weighted quick-union: Java implementation

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

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

**Union.** Modify quick-union to:
- Link root of smaller tree to root of larger tree.
- Update the \( sz[] \) array.

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

**Running time.**
- Find: takes time proportional to depth of $p$.
- Union: takes constant time, given roots.

**Proposition.** Depth of any node $x$ is at most $\lg N$.

$N = 10$
\[
\text{depth}(x) = 3 \leq \lg N
\]
Weighted quick-union analysis

Running time.
- Find: takes time proportional to depth of $p$.
- Union: takes constant time, given roots.

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

Pf. What causes the depth of object $x$ to 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?

\[ \lg = \text{base-2 logarithm} \]
# Weighted quick-union analysis

## Running time.
- Find: takes time proportional to depth of \( p \).
- Union: takes constant time, given roots.

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>( N \dagger )</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>weighted QU</td>
<td>N</td>
<td>( \lg N \dagger )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
</tr>
</tbody>
</table>

\( \dagger \) includes cost of finding roots

**Time/space tradeoff:** Weighted QU uses \( O(N) \) more space to improve running time.

**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 $\text{id}[]$ of each examined node to point to that root.
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.
**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.
**Improvement 2: path compression**

Quick union with path compression. Just after computing the root of \( p \), set the \( \text{id[]} \) of each examined node to point to that root.
**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.

![Diagram of Quick Union with Path Compression](image)

**Bottom line.** Now, `find()` has the side effect of compressing the tree.
Path compression: Java implementation

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

**Simpler one-pass variant (path halving):** Make 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.
Weighted quick-union with path compression: amortized analysis

**Proposition.** [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of $M$ union–find ops on $N$ objects makes $\leq c(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M \alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\lg^* N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>$65536$</td>
<td>4</td>
</tr>
<tr>
<td>$2^{65536}$</td>
<td>5</td>
</tr>
</tbody>
</table>

Linear-time algorithm for $M$ union-find ops on $N$ objects?

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

**Amazing fact.** [Fredman-Saks] No linear-time algorithm exists. in "cell-probe" model of computation
Summary

**Key point.** Weighted quick union (and/or path compression) 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 N )</td>
</tr>
<tr>
<td>quick-union</td>
<td>( M N )</td>
</tr>
<tr>
<td>weighted QU</td>
<td>( N + M \log N )</td>
</tr>
<tr>
<td>QU + path compression</td>
<td>( N + M \log N )</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>( N + M \operatorname{lg^*} N )</td>
</tr>
</tbody>
</table>

order of growth for \( M \) union–find operations on a set of \( N \) objects

**Ex.** [\(10^9\) unions and finds with \(10^9\) objects]

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

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Union-find applications

- Percolation.
- Games (Go, Hex).

✓ Dynamic 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.
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.
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 \textit{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
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 $p^*$ ?

![Diagram of percolation phase transition with threshold $p^*$ at $0.593$]
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 $p^*$.
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 and use union-find.
Dynamic connectivity solution to estimate percolation threshold

**Q.** How to check whether an $N$-by-$N$ system percolates?
- Create an object for each site and name them 0 to $N^2 - 1$. 

![Diagram](image-url)
Q. How to check whether an $N$-by-$N$ system percolates?

- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component iff connected by open sites.
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an $N$-by-$N$ system percolates?
- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component iff connected by open sites.
- Percolates iff any site on bottom row is connected to any site on top row.

brute-force algorithm: $N^2$ calls to connected()

$N = 5$

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

Clever trick. Introduce 2 virtual sites (and connections to top and bottom).

- Percolates iff virtual top site is connected to virtual bottom site.

More efficient algorithm: only 1 call to connected()
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

\[
N = 5
\]

open this site

open site

blocked site
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?
A. Mark new site as open; connect it to all of its adjacent open sites.

open this site

\[ N = 5 \]

open site

blocked site

up to 4 calls to union()
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.