Slides from Kevin Wayne on Union-Find and Percolation
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.
dynamic connectivity
quick find
quick union
improvements
applications
Dynamic connectivity

Given a set of objects

- **Union**: connect two objects.
- **Connected**: is there a path connecting the two objects?

\[
\begin{align*}
\text{union}(3, 4) \\
\text{union}(8, 0) \\
\text{union}(2, 3) \\
\text{union}(5, 6) \\
\text{connected}(0, 2) & \quad \text{no} \\
\text{connected}(2, 4) & \quad \text{yes} \\
\text{union}(5, 1) \\
\text{union}(7, 3) \\
\text{union}(1, 6) \\
\text{union}(4, 8) \\
\text{connected}(0, 2) & \quad \text{yes} \\
\text{connected}(2, 4) & \quad \text{yes}
\end{align*}
\]
Q. Is there a path from p to q?

A. Yes.
Dynamic connectivity applications involve manipulating objects of all types.

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

When programming, convenient to name sites 0 to N-1.

- Use integers as array index.
- Suppress details not relevant to union-find.

Can use symbol table to translate from site names to integers: stay tuned (Chapter 3)
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 components. Maximal set of objects that are mutually connected.
Implementing the operations

**Find query.** Check if two objects are in the same component.

**Union command.** Replace components containing two objects with their union.

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

\[
\begin{align*}
\{0\} & \quad \{1\} & \quad \{2\} & \quad \{3\} & \quad \{4\} & \quad \{5\} & \quad \{6\} & \quad \{7\} \\
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7
\end{align*}
\]

3 connected components

\[
\begin{align*}
\{0\} & \quad \{1 2 3 4 5 6 7\}
\end{align*}
\]

2 connected components
*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.

**Union-find data type (API)**

```java
public class UF {
    UF(int N)
        initialize union-find data structure with $N$ objects (0 to N-1)
    void union(int p, int q)
        add connection between p and q
    boolean connected(int p, int q)
        are p and q in the same component?
    int find(int p)
        component identifier for p (0 to N-1)
    int count()
        number of components
}
```
Dynamic-connectivity client

• Read in number of objects $N$ from standard input.
• Repeat:
  - read in pair of integers from standard input
  - write out pair if they are not already connected

```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)) continue;
        uf.union(p, q);
        StdOut.println(p + " " + q);
    }
}
```

% more tiny.txt

10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
dynamic connectivity
quick find
quick union
improvements
applications
Quick-find [eager approach]

Data structure.
- Integer array \( \text{id[]} \) of size \( N \).
- Interpretation: \( p \) and \( q \) in same component iff they have the same id.

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

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

Data structure.
- Integer array $\text{id}[]$ of size $N$.
- Interpretation: $p$ and $q$ in same component iff 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>$\text{id}[i]$</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>

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

- $\text{id}[3] = 9$; $\text{id}[6] = 6$
- 3 and 6 in different components
- 5 and 6 are connected
- 2, 3, 4, and 9 are connected
Data structure.
• Integer array $id[]$ of size $N$.
• Interpretation: $p$ and $q$ in same component iff they have the same id.

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

Union. To merge sets containing $p$ and $q$, change all entries with $id[p]$ to $id[q]$.
Quick-find example

id[]

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

id[p] and id[q] differ, so union() changes entries equal to id[p] to id[q] (in red)

id[p] and id[q] match, so no change
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 boolean connected(int p, int q)
    {  return id[p] == id[q];  }

    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;
    }
}
Quick-find is too slow

Cost model. Number of array accesses (for read or write).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>init</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>

Quick-find defect.

- Union too expensive.
- Trees are flat, but too expensive to keep them flat.
- Ex. Takes $N^2$ array accesses to process sequence of $N$ union commands on $N$ objects.
Data structure.

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

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

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

3's root is 9; 5's root is 6

keep going until it doesn't change
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.

<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>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

3's root is 9; 5's root is 6
3 and 5 are in different components
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 example

```
ip  q
4  3
3  8
6  5
9  4
2  1
```

```
 id[]
0  1  2  3  4  5  6  7  8  9
0  1  2  3  4  5  6  7  8  9
0  1  2  3  6  7  8  9
0  1  2  3  6  7  8  9
0  1  2  3  5  6  7  8  9
0  1  2  3  5  6  7  8  9
0  1  2  3  5  6  7  8
0  1  2  3  5  6  7  8
0  1  2  3  5  6  7
0  1  2  3  5  6
0  1  2  3  5
```
Quick-union example

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

8 9 0 1 1 0 8 3 5 5 7 8 8
5 0 0 1 1 0 8 3 5 5 7 8 8
 0 1 1 8 3 0 5 7 8 8
7 2 0 1 1 0 3 0 5 7 8 8
 0 1 1 0 3 0 5 1 8 8
6 1 0 1 1 0 3 0 5 1 8 8
 1 1 1 8 3 0 5 1 8 8
1 0 1 1 1 8 3 0 5 1 8 8
6 7 1 1 1 8 3 0 5 1 8 8
Quick-union: Java implementation

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

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

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

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

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

- Set id of each object to itself (N array accesses)
- Chase parent pointers until reach root (depth of i array accesses)
- Check if p and q have same root (depth of p and q array accesses)
- Change root of p to point to root of q (depth of p and q array accesses)
Quick-union is also too slow

**Cost model.** Number of array accesses (for read or write).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>init</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>

† includes cost of finding root

**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).
› 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 small tree below large one.
Quick-union and weighted quick-union example

Quick-union and weighted quick-union (100 sites, 88 union() operations)
Data structure. Same as quick-union, but maintain extra array sz[i] to count number of objects in the tree rooted at i.

Find. Identical to quick-union.

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

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

```java
int i = root(p);
int j = root(q);
if (sz[i] < sz[j]) {
    id[i] = j;
    sz[j] += sz[i];
} else {
    id[j] = i;
    sz[i] += sz[j];
}```
Running time.
• 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 $\lg N$. 

$N = 10$
$\text{depth}(x) = 3 \delta \lg N$
Running time.
• 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?
Running time.
- 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 \( \lg N \).

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

1 linked to 6 because of path compression

7 linked to 6 because of path compression
Proposition. Starting from an empty data structure, any sequence of \( M \) union–find operations on \( N \) objects makes at most proportional to \( N + M \lg^* N \) array accesses.

- Proof is very difficult.
- Can be improved to \( N + M \langle (M, N) \rangle \).
- But the algorithm is still simple!

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. No linear-time algorithm exists.
**Bottom line.** WQUPC 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 lg* N</td>
</tr>
</tbody>
</table>

**M union–find operations on a set of N objects**

**Ex.** [10⁹ unions and finds with 10⁹ objects]
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.
› dynamic connectivity
› quick find
› quick union
› improvements
› applications
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 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>
Depends on site vacancy probability $p$.

- $p$ low (0.4) does not percolate
- $p$ medium (0.6) percolates?
- $p$ high (0.8) 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 $p^*$.
Q. How to check whether an $N$-by-$N$ system percolates?

*Dynamic connectivity solution to estimate percolation threshold*

$N = 5$
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 of an $N$-by-$N$ system with open and blocked sites]
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 set if connected by open sites.
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 set if connected by open sites.
• Percolates iff any site on bottom row is connected to site on top row.

Dynamic connectivity solution to estimate percolation threshold

brute-force algorithm: $N^2$ calls to connected()
Clever trick. Introduce two virtual sites (and connections 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
**Clever trick.** Introduce two virtual sites (and connections to top and bottom).

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

---

**Diagram:**

- **N = 5**
- **empty open site** (not connected to top)
- **full open site** (connected to top)
- **blocked site**
- **virtual top site**
- **virtual bottom site**
Q. How to model as dynamic connectivity problem when opening a new site?
Q. How to model as dynamic connectivity problem when opening a new site?
A. Connect new site to all of its adjacent open sites.

\( N = 5 \)

up to 4 calls to union()
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.