



<http://algs4.cs.princeton.edu>

## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*

# Subtext of today's lecture (and this course)

---

## 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.



## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*

# Dynamic connectivity

---

Given a set of N objects.

- **Union command:** connect two objects.
- **Find/connected query:** is there a path connecting the two objects?

`union(4, 3)`

`union(3, 8)`

`union(6, 5)`

`union(9, 4)`

`union(2, 1)`

`connected(0, 7)` ✗

`connected(8, 9)` ✓

`union(5, 0)`

`union(7, 2)`

`union(6, 1)`

`union(1, 0)`

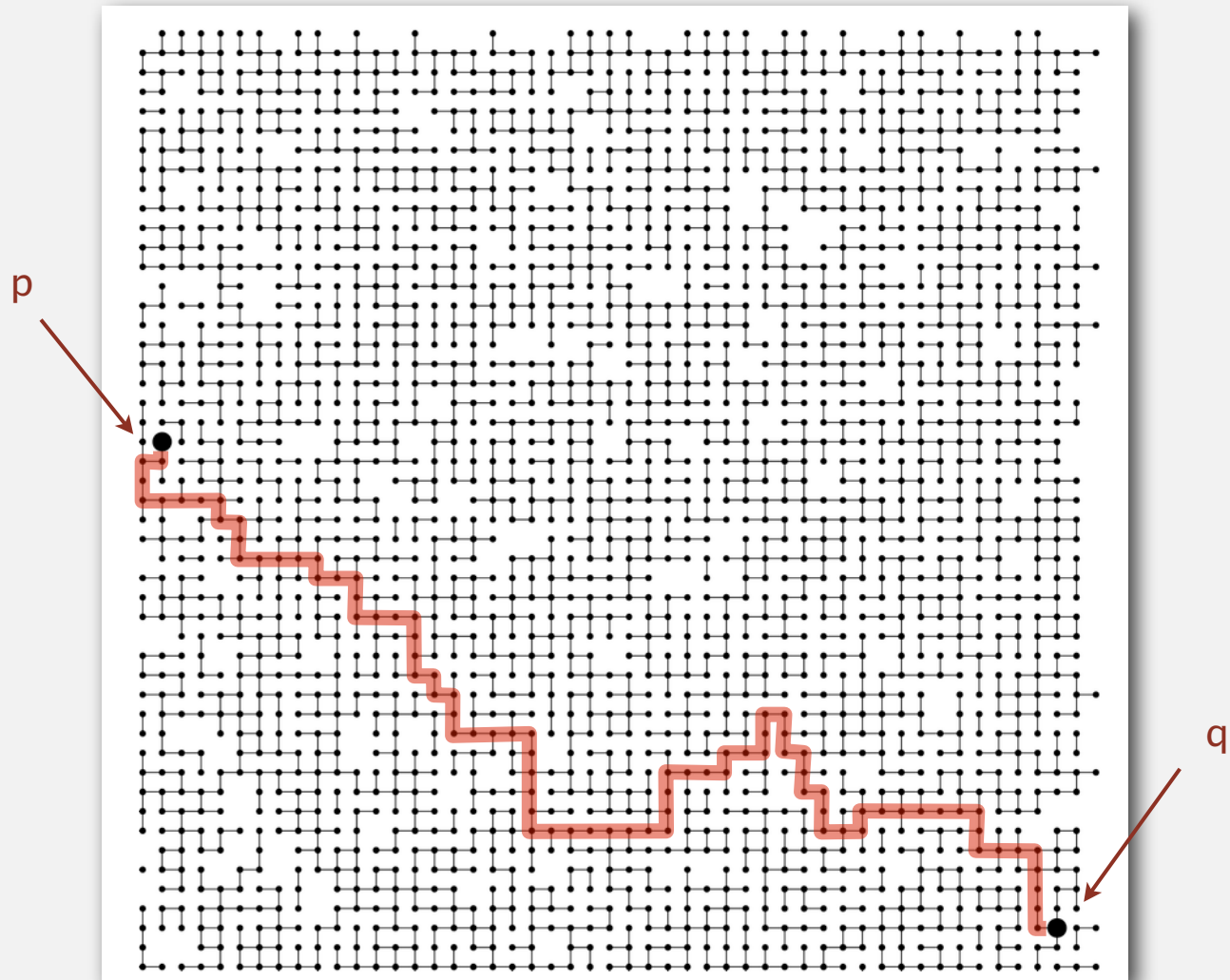
`connected(0, 7)` ✓



## Connectivity example

---

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



A. Yes.

# Modeling the objects

---

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 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.



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



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

3 connected components

# Implementing the operations

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

**Union command.** Replace components 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.

```
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
  - if they are not yet connected, connect them and print out pair

```
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
```



## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*



## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*

# Quick-find [eager approach]

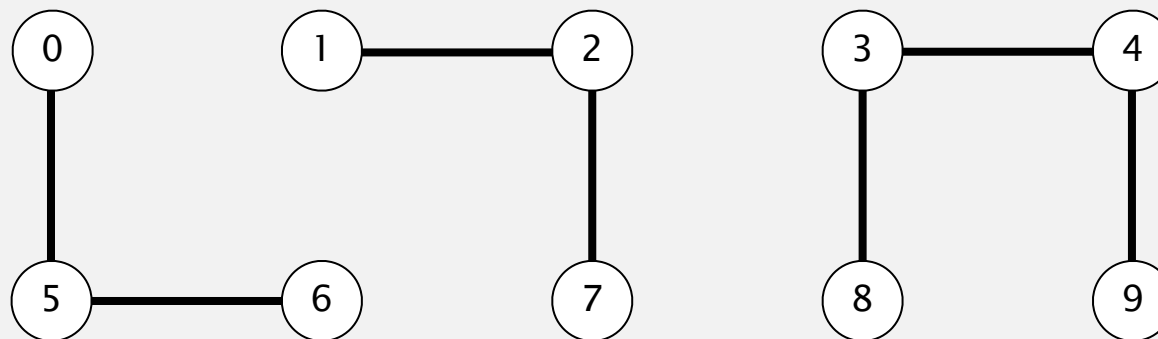
## Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `p` and `q` are connected iff they have the same `id`.

if and only if  
↙

|                   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|
|                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <code>id[]</code> | 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: `p` and `q` are connected iff they have the same `id`.

|                   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|
|                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <code>id[]</code> | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 1 | 8 | 8 |

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

`id[6] = 0; id[1] = 1`  
6 and 1 are not connected

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

|                   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|
|                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <code>id[]</code> | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |



problem: many values can change

after union of 6 and 1

# Quick-find demo

---



0

1

2

3

4

5

6

7

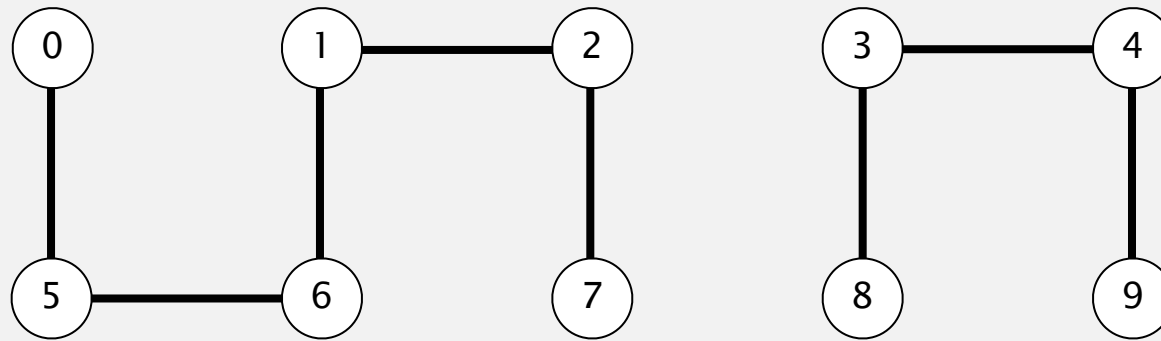
8

9

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

# Quick-find demo

---



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



# Quick-find: Java implementation

---

```
public class QuickFindUF
{
```

```
    private int[] id;
```

```
    public QuickFindUF(int N)
    {
```

```
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
```

← set id of each object to itself  
(N array accesses)

```
    }
```

```
    public boolean connected(int p, int q)
    { return id[p] == id[q]; }
```

← check whether p and q  
are in the same component  
(2 array accesses)

```
    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;
```

← 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).

| algorithm  | initialize | union | find |
|------------|------------|-------|------|
| quick-find | $N$        | $N$   | 1    |

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 commands on  $N$  objects.

quadratic



# 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.

a truism (roughly)  
since 1950!



## 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  $\Rightarrow$  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*



## 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[...id[i]...]]]`.

keep going until it doesn't change  
(algorithm ensures no cycles)

|                   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|
|                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <code>id[]</code> | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |



# 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]...]]]`.

|                   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|
|                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <code>id[]</code> | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |

**Find.** Check if `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.

|                   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|
|                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <code>id[]</code> | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 6 |

↑  
only one value changes



root of 3 is 9  
root of 5 is 6  
3 and 5 are not connected



# Quick-union demo

---



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



# Quick-union demo

---



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

# Quick-union: Java implementation

---

```
public class QuickUnionUF
{
```

```
    private int[] id;
```

```
    public QuickUnionUF(int N)
    {
```

```
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
```

← set id of each object to itself  
(N array accesses)

```
    }
```

```
    private int root(int i)
    {
```

```
        while (i != id[i]) i = id[i];
        return i;
```

← chase parent pointers until reach root  
(depth of i array accesses)

```
    }
```

```
    public boolean connected(int p, int q)
    {
```

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

← check if p and q have same root  
(depth of p and q array accesses)

```
    }
```

```
    public void union(int p, int q)
    {
```

```
        int i = root(p);
        int j = root(q);
        id[i] = j;
```

← 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).

| algorithm   | initialize | union       | find |
|-------------|------------|-------------|------|
| quick-find  | $N$        | $N$         | 1    |
| quick-union | $N$        | $N^\dagger$ | $N$  |

← worst case

$\dagger$  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 too expensive (could be  $N$  array accesses).



## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*



## 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.

quick-union



reasonable alternatives:  
union by height or "rank"

weighted



# Weighted quick-union demo

---



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

# Weighted quick-union demo

---



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



# Quick-union and weighted quick-union example

quick-union



*average distance to root: 5.11*

weighted



*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.** Identical to quick-union.

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

**Union.** Modify quick-union to:

- Link root of smaller tree to root of larger tree.
- Update the `sz[]` array.

```
int i = root(p);
int j = root(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  $p$  and  $q$ .
- Union: takes constant time, given roots.

$\lg$  = base-2 logarithm

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



# Weighted quick-union analysis

---

## 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?



# Weighted quick-union analysis

---

## 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  $\lg N$ .

| algorithm   | initialize | union           | connected |
|-------------|------------|-----------------|-----------|
| quick-find  | $N$        | $N$             | 1         |
| quick-union | $N$        | $N^\dagger$     | $N$       |
| weighted QU | $N$        | $\lg N^\dagger$ | $\lg N$   |

$\dagger$  includes cost of finding roots

**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.



## 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 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.



## Path compression: Java implementation

---

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

**Simpler one-pass variant:** Make every other node in path point to its grandparent (thereby halving path length).

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

← only one extra line of code !

**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.


| N           | $\lg^* N$ |
|-------------|-----------|
| 1           | 0         |
| 2           | 1         |
| 4           | 2         |
| 16          | 3         |
| 65536       | 4         |
| $2^{65536}$ | 5         |

iterate log function

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

---

**Bottom line.** Weighted quick union (with path compression) makes it possible to solve problems that could not otherwise be addressed.

| algorithm                      | worst-case time |
|--------------------------------|-----------------|
| quick-find                     | $M N$           |
| quick-union                    | $M N$           |
| weighted QU                    | $N + M \log N$  |
| QU + path compression          | $N + M \log N$  |
| weighted QU + path compression | $N + M \lg^* N$ |

**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*



## 1.5 UNION-FIND

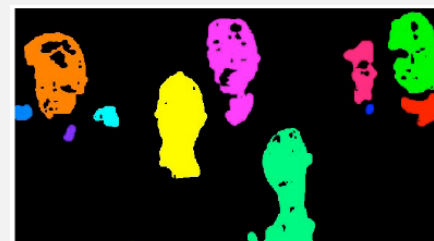
---

- ▶ *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.





# 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** iff top and bottom are connected by open sites.



*does not percolate*



# 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** iff top and bottom are connected by open sites.

| model              | system     | vacant site | occupied site | percolates   |
|--------------------|------------|-------------|---------------|--------------|
| electricity        | material   | conductor   | insulated     | conducts     |
| fluid flow         | material   | empty       | blocked       | porous       |
| social interaction | population | person      | empty         | communicates |

# Likelihood of percolation

---

Depends on 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  $p^*$  ?



# Monte Carlo simulation

---

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



$N = 20$



# Dynamic connectivity solution to estimate percolation threshold

---

Q. How to check whether an  $N$ -by- $N$  system percolates?

$N = 5$



open site



blocked site

# Dynamic connectivity solution to estimate percolation threshold

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

$N = 5$



# 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 if connected by open sites.

$N = 5$





# 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 if connected by open sites.
- Percolates iff any site on bottom row is connected to site on top row.

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

$N = 5$



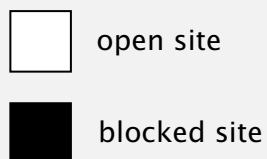
# Dynamic connectivity solution to estimate percolation threshold

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

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

efficient algorithm: only 1 call to `connected()`

$N = 5$



# Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new 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.



# Percolation threshold

Q. What is percolation threshold  $p^*$  ?

A. About 0.592746 for large square lattices.

constant known only via simulation



Fast algorithm **enables** accurate answer to scientific question.



## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*

# Subtext of today's lecture (and this course)

---

## 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.



<http://algs4.cs.princeton.edu>

## 1.5 UNION-FIND

---

- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*