Introduction to Algorithms

Lecture 1:Union Find Problem

Yao-Chung Fan
yfan@nchu.edu.tw

The Goal of This Course...

Steps to developing a usable algorithm.

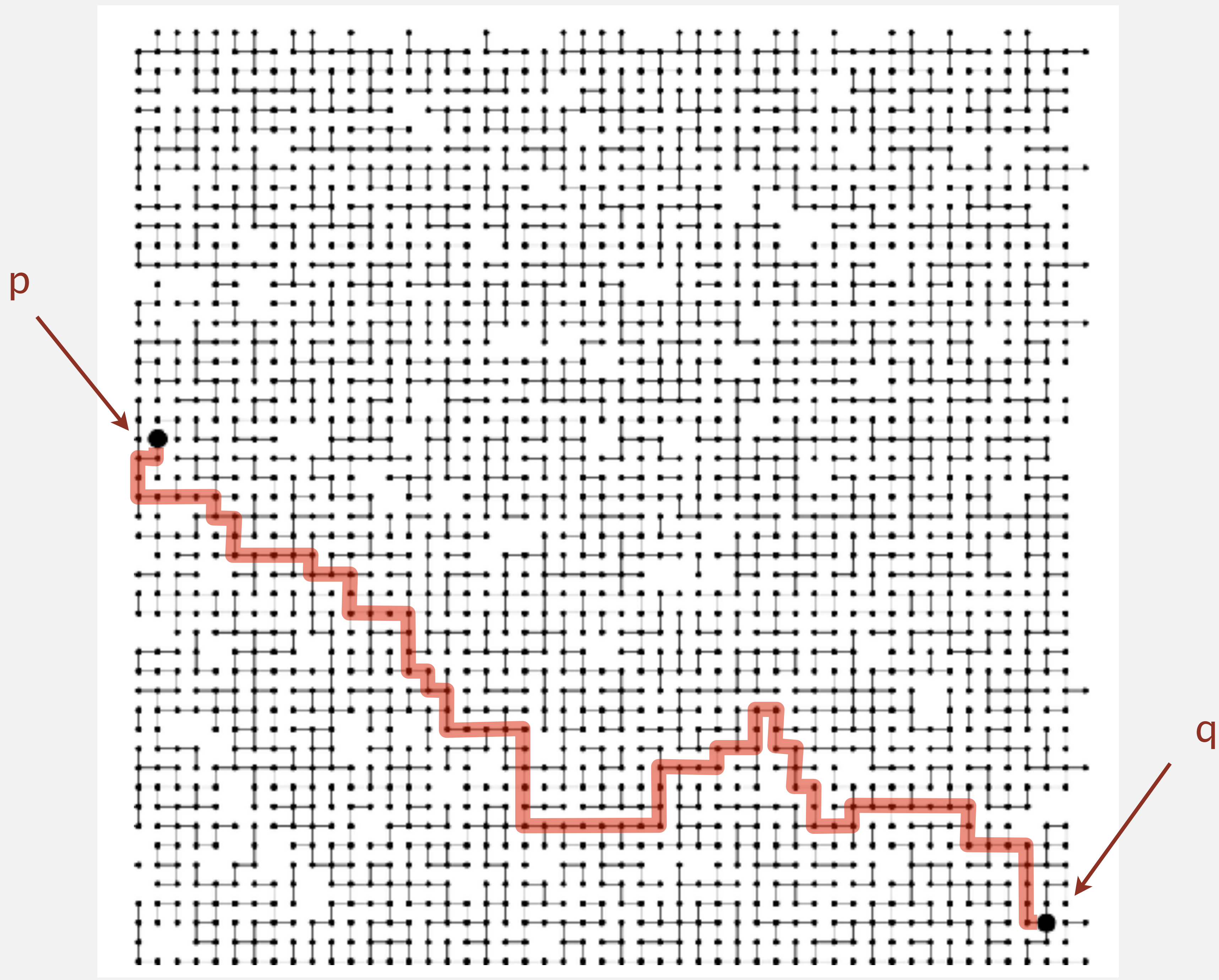
- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why not.
- Find a way to address the problem.
- Iterate until satisfied.

UNION-FIND ALGORITHM

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

A larger connectivity example

Q. Is there a path connecting p and q ?



A. Yes.

Dynamic connectivity problem

Given a set of N objects, support two operations:

- 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? ✔

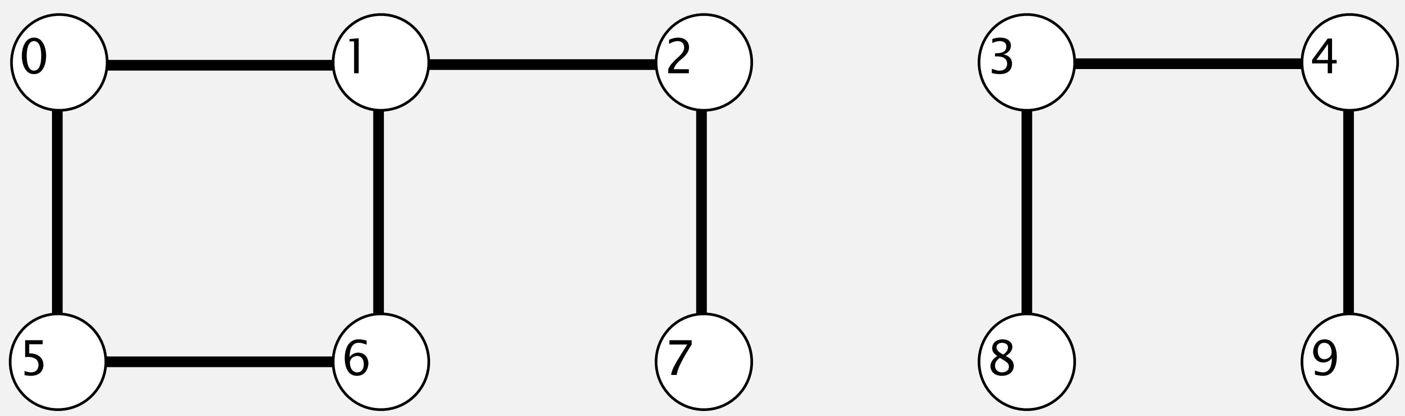
connect 5 and 0

connect 7 and 2

connect 6 and 1

connect 1 and 0

are 0 and 7 connected? ✔



Modeling the objects

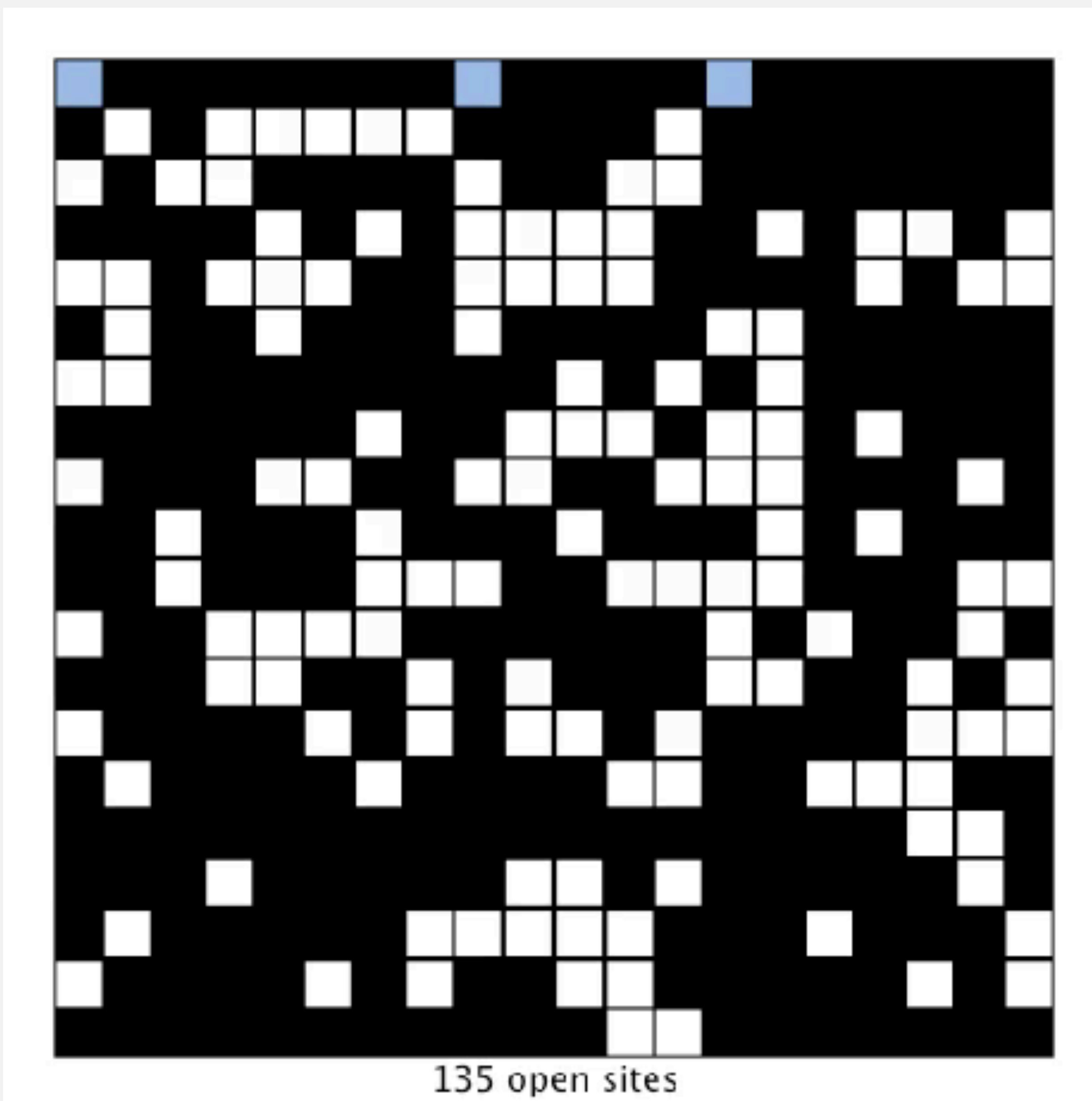
Applications involve manipulating objects of all types.

- Computers in a network.
- Friends in a social network.
- Transistors in a computer chip.
- Virus Diffusion in a social network

病毒如何傳遞？多久你會變成一個殭屍？

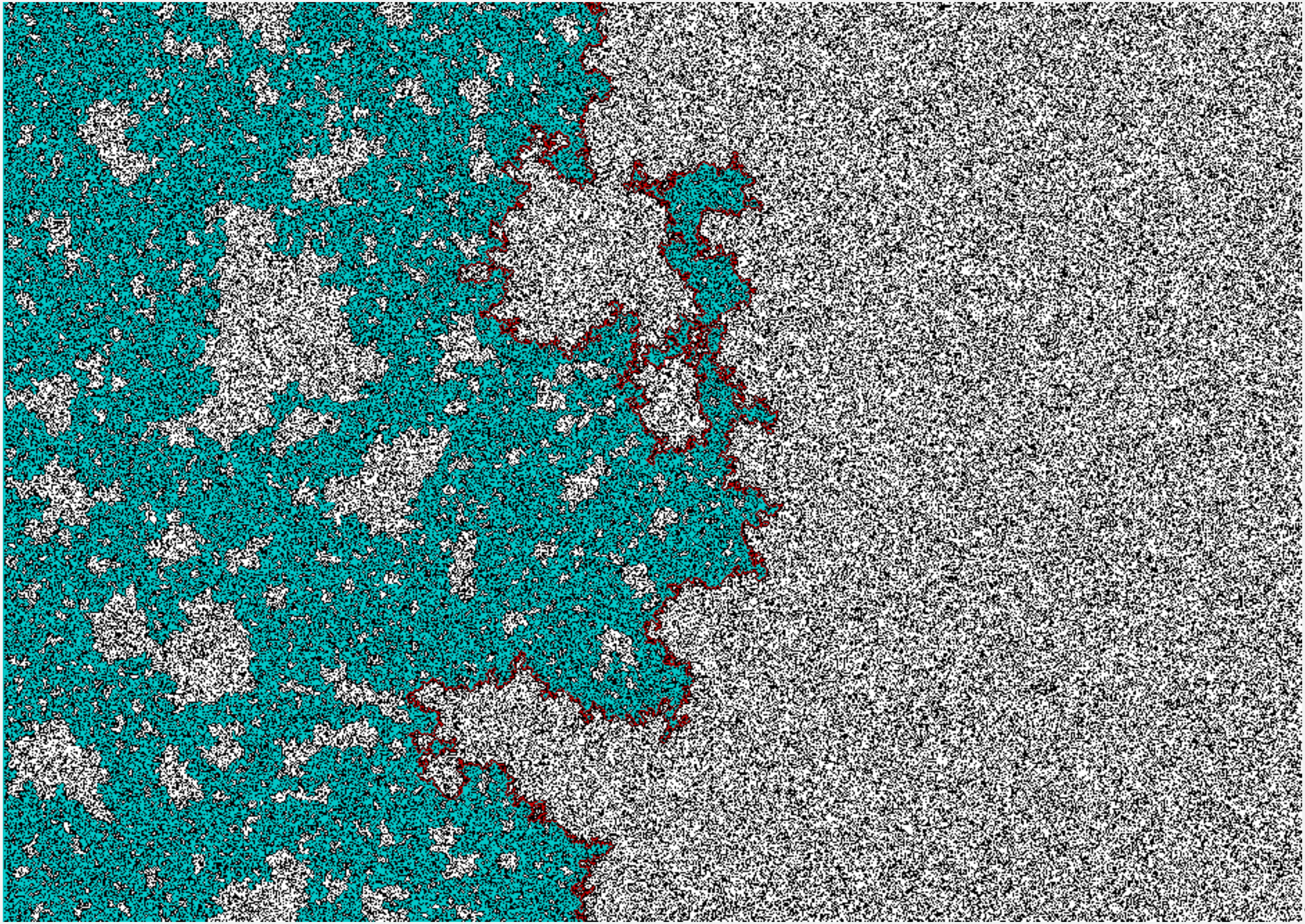
Monte Carlo simulation

- 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^*.



$N = 20$

Percolation 浸透



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.**

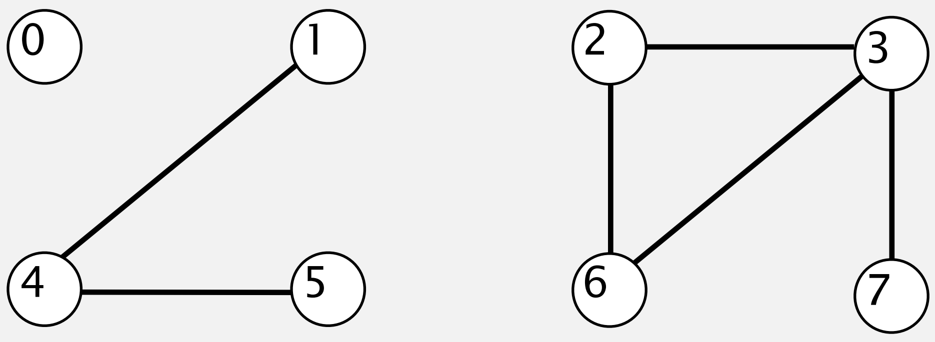
public class UF		
	UF(int N)	*initialize union-find data structure with N singleton objects (0 to N – 1)*
void	union(int p, int q)	*add connection between p and q*
boolean	connected(int p, int q)	*are p and q in the same component?*

Union-Find Algorithm

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

Connected Component

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



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

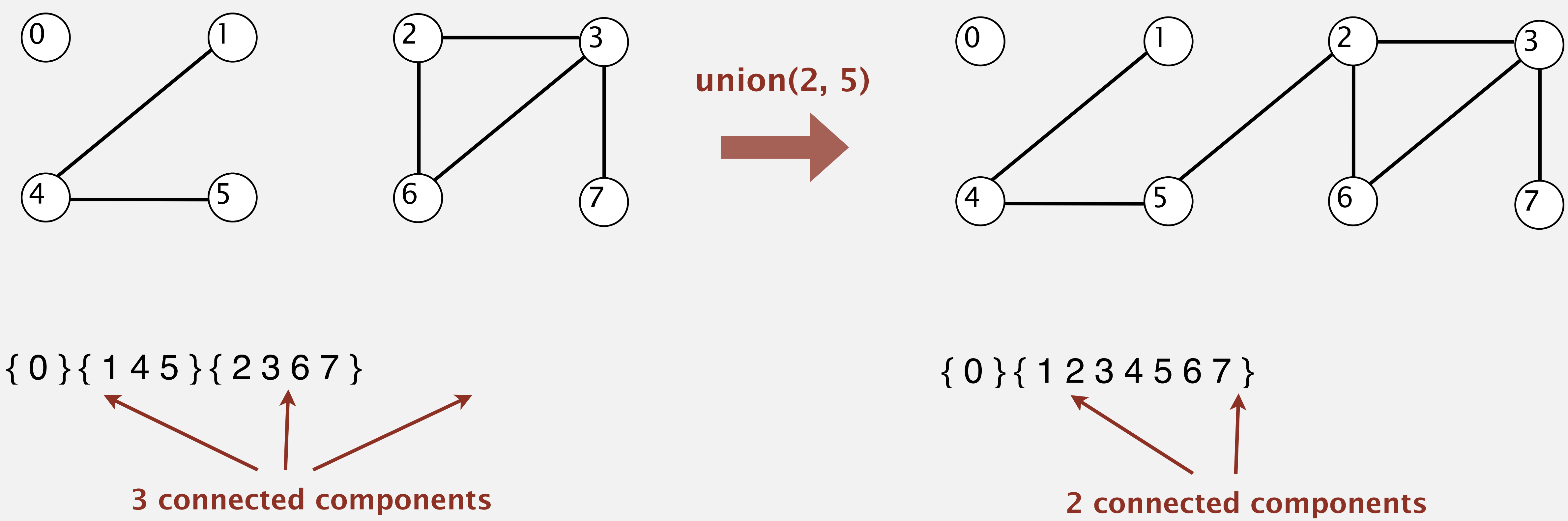
3 connected components

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.



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.

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?*

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

1-line implementation of connected()

Quick-find [eager approach]

Data structure.

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

if and only if

	0	1	2	3	4	5	6	7	8	9
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.

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

id[6] = 0; id[1] = 1

6 and 1 are not connected

* **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].

union (6, 1)

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

after union of 6 and 1

find examines id[5] *and* id[9]

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

union has to change all 1s to 8s

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

Quick-find overview

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;
   }

   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).

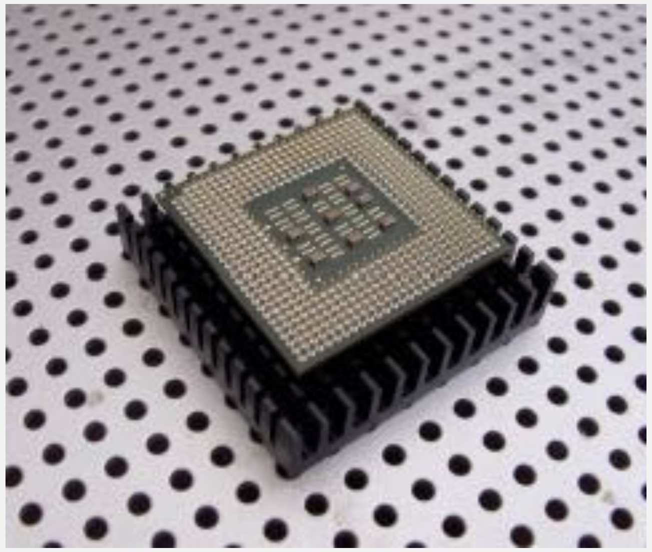
algorithm	initialize	union	find	connected
quick-find	N	N	1	1

order of growth of number of array accesses

quadratic

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 but even worse



Rough standard (for now).

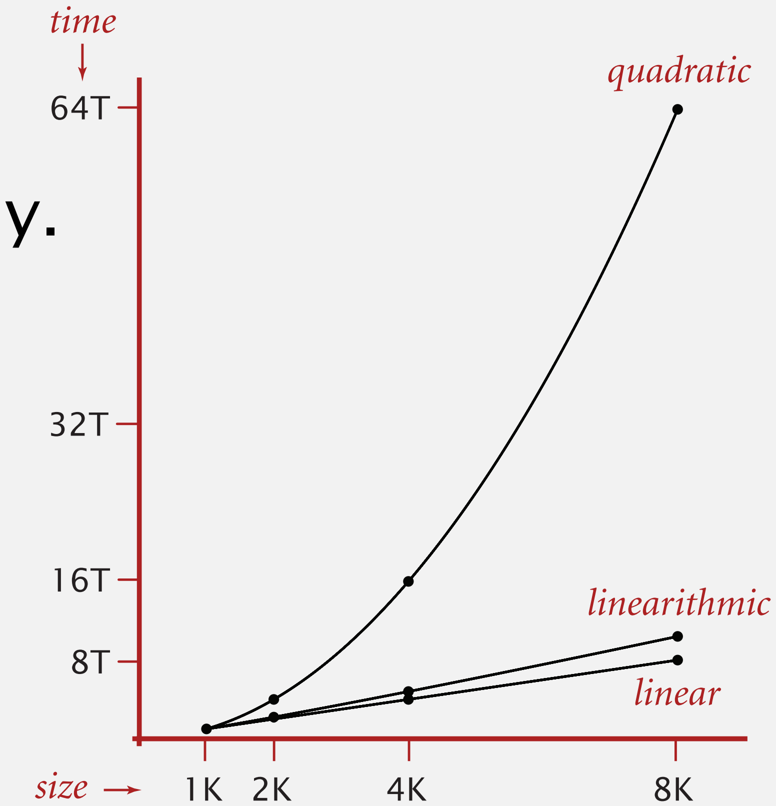
- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

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!



NEED FOR SPEED, OTHERWISE?

Take a Rest







Take a Rest

Natural Selection

Evolution

The Goal of 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 not.
- Find a way to address the problem.
- Iterate until satisfied.

Union-Find Problem

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

What is the problem with Quick-Find?

```
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;
}
```

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

Problem: many values can change

Quick-union [lazy approach]

Data structure.

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

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

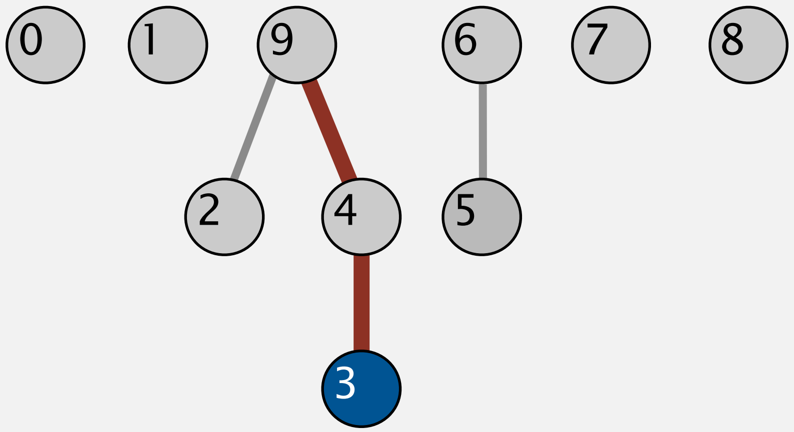
Quick-union [lazy approach]

Data structure.

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

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

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



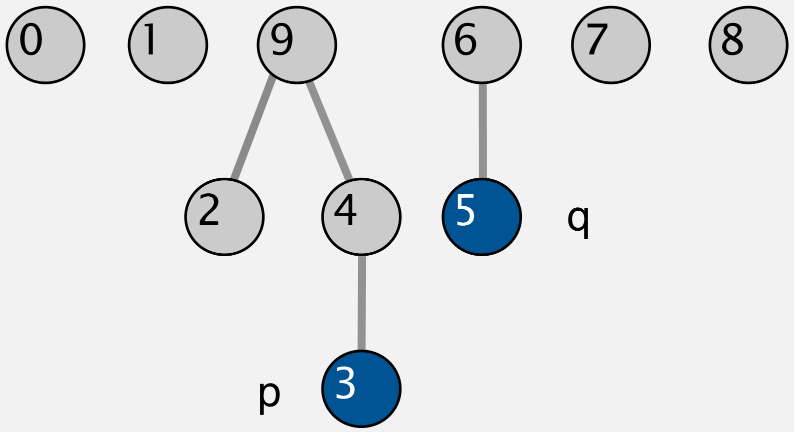
parent of 3 is 4

root of 3 is 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]...]]].



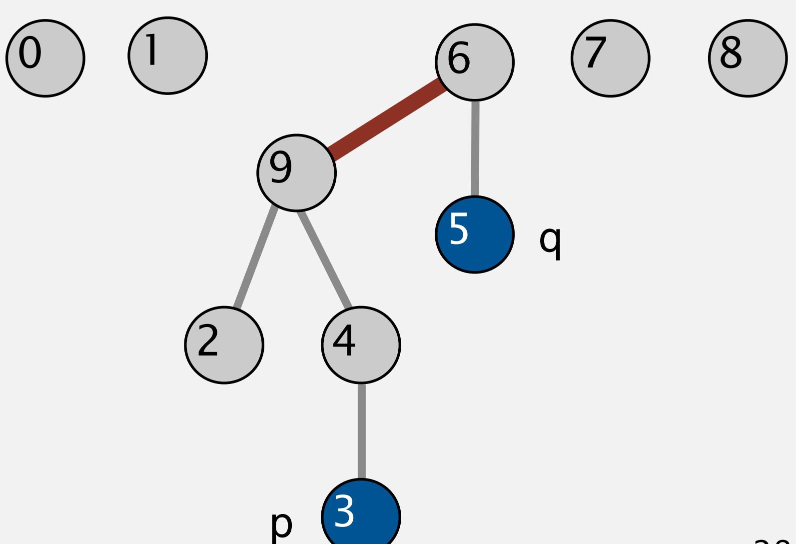
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9

Find. What is the root of p?

Connected. Do p and q have the same root?

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

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
id[]	0	1	9	4	9	6	6	7	8	6

only one value changes

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;
   }

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


   public void union(int p, int q)
   {
      int proot = find(p);
      int qroot = find(q);
      id[proot] = qroot;
   }
}
```

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

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

change root of p to point to root of q
(depth of p and q array accesses)

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;
   }

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


   public void union(int p, int q)
   {
      int proot = find(p);
      int qroot = find(q);
      id[proot] = qroot;
   }
}
```

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

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

change root of p to point to root of q
(depth of p and q array accesses)

隨堂小考1:

```
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)
   {
     id[p] = q;
   }
}
```

請實際trace一遍

改成這樣，發生什麼事，有什麼優缺點？

Code Comparison

```
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;
   }
}
```

```
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 proot = find(p);
      int qroot = find(q);
      id[proot] = qroot;
   }
}
```

QuickUnion is the solution ? (期中考題 01 12 23 34)

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	Tree Height	Tree Height	Tree Height

想想看：
Please give an example to show the worst case of the quick union…

練習: 請舉一個Quick-Union最好的case

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	Tree Height	Tree Height	Tree Height

Please give an example to show the **best** case of the quick union…

Quick-union is also too slow

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

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N †	N	N

worst case

† 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).

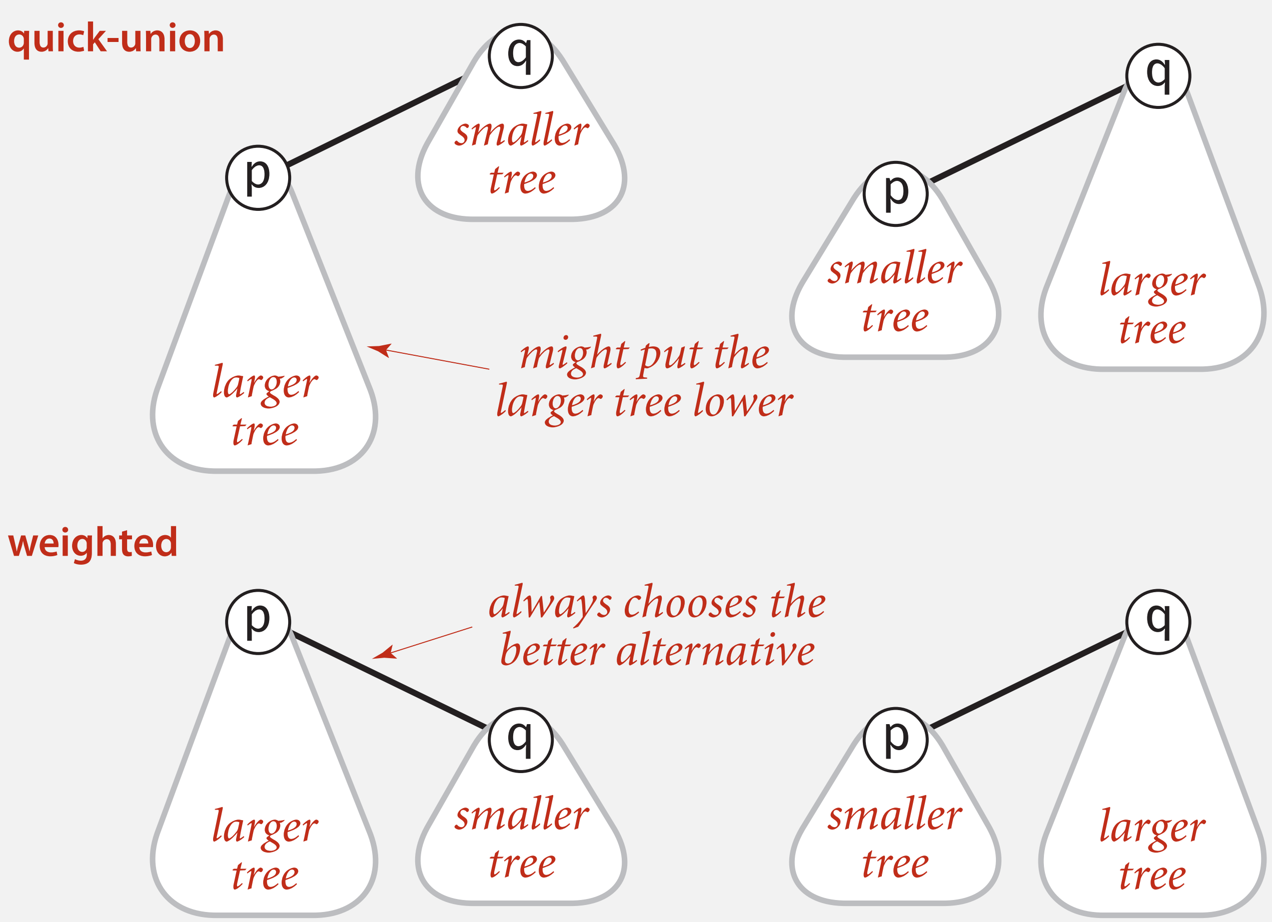
UNION-FIND PROBLEM

- *dynamic connectivity*
- *quick find*
- *quick union*
- *weighted quick union*
- *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.




```
public void union(int p, int q)
{
    int proot = find(p);
    int qroot = find(q);
    id[proot] = qroot;
}
```

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.

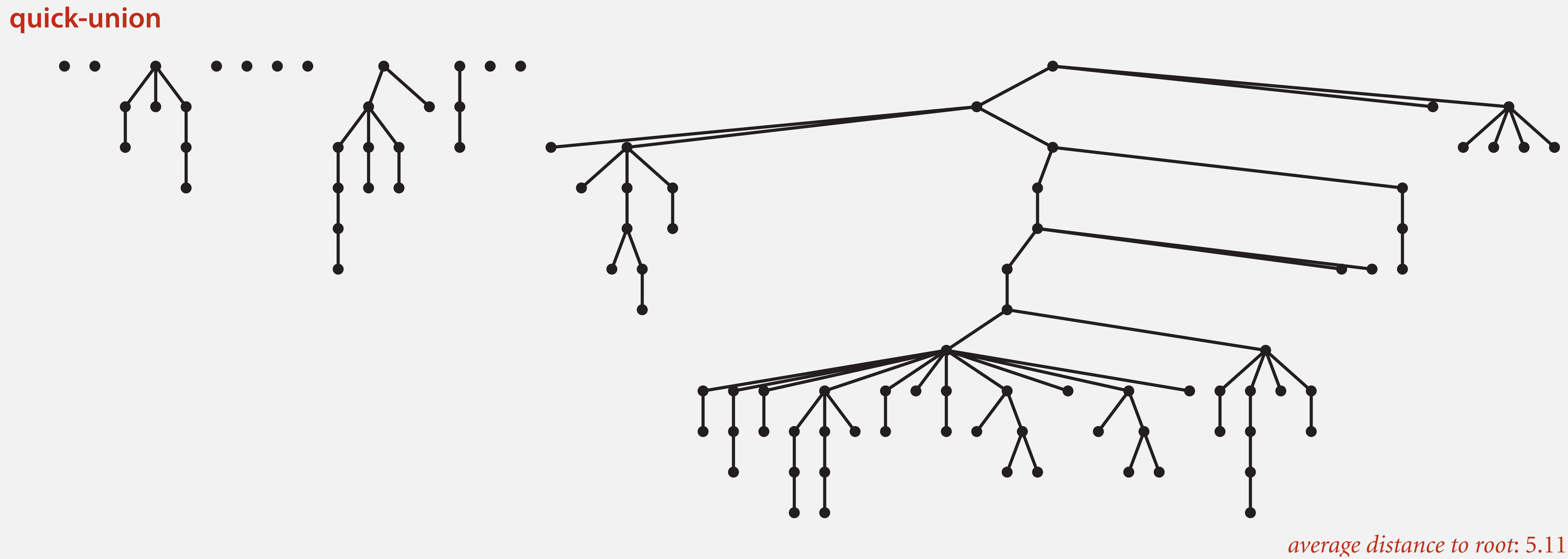
Union. Modify quick-union to:

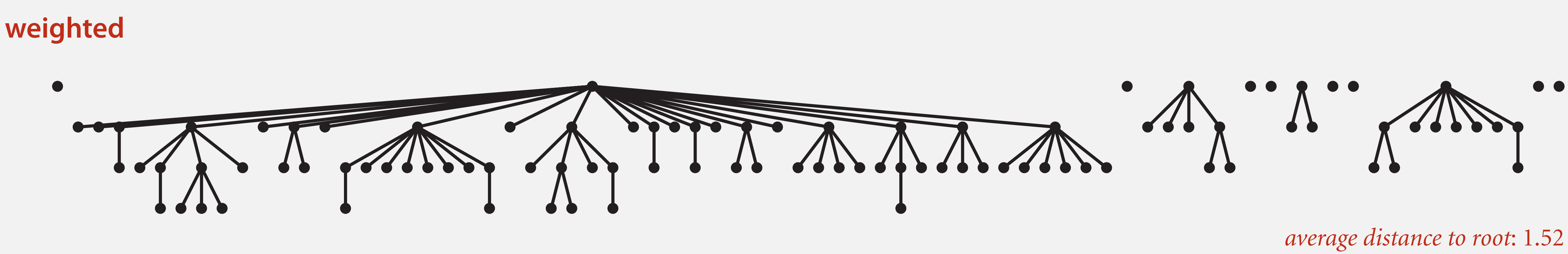
- Link root of smaller tree to root of larger tree.
- Update the sz[] array.

```
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]; }
```

Quick-union and weighted quick-union example





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

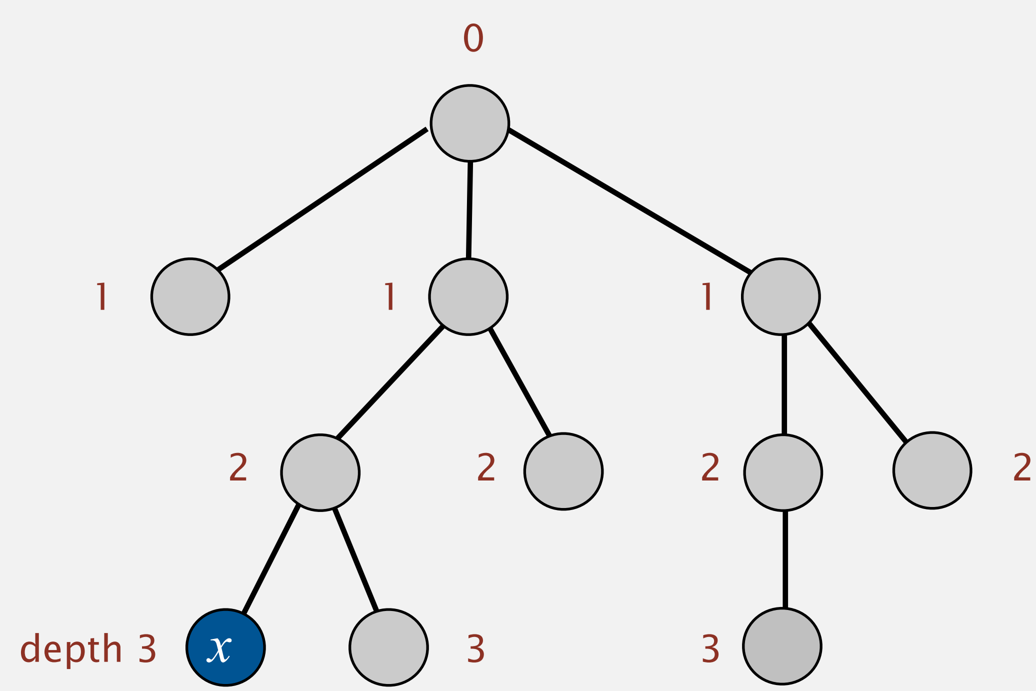
Weighted quick-union analysis

Running time.

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

lg = base-2 logarithm

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



N = 11
depth(x) = 3 ≤ lg N

Weighted quick-union analysis

Running time.

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

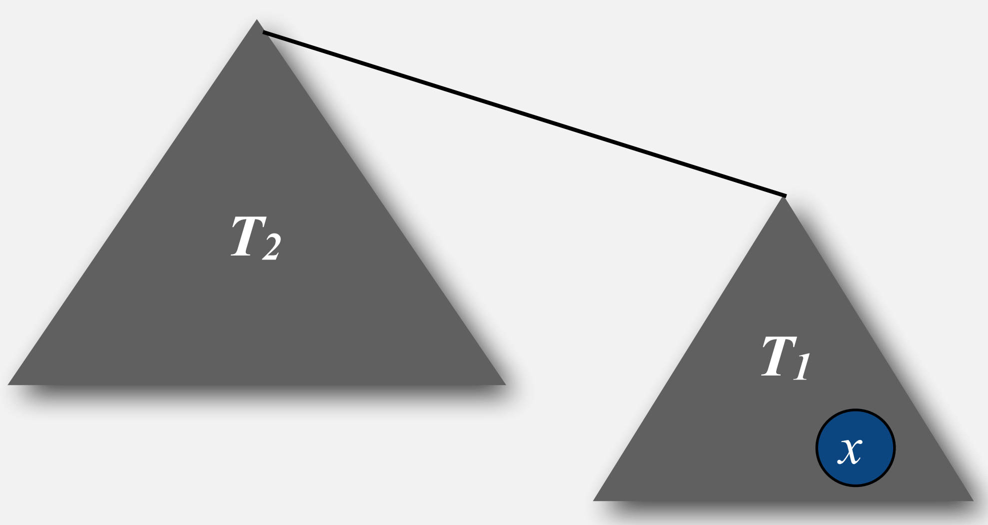
lg = base-2 logarithm

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?



$$\left.\begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \\ 16 \\ \vdots \\ N \end{array}\right\} \lg N$$

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 $\lg N$.

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N †	N	N
weighted QU	N	lg N †	lg N	lg N

† includes cost of finding roots

Q. Stop at guaranteed acceptable performance?

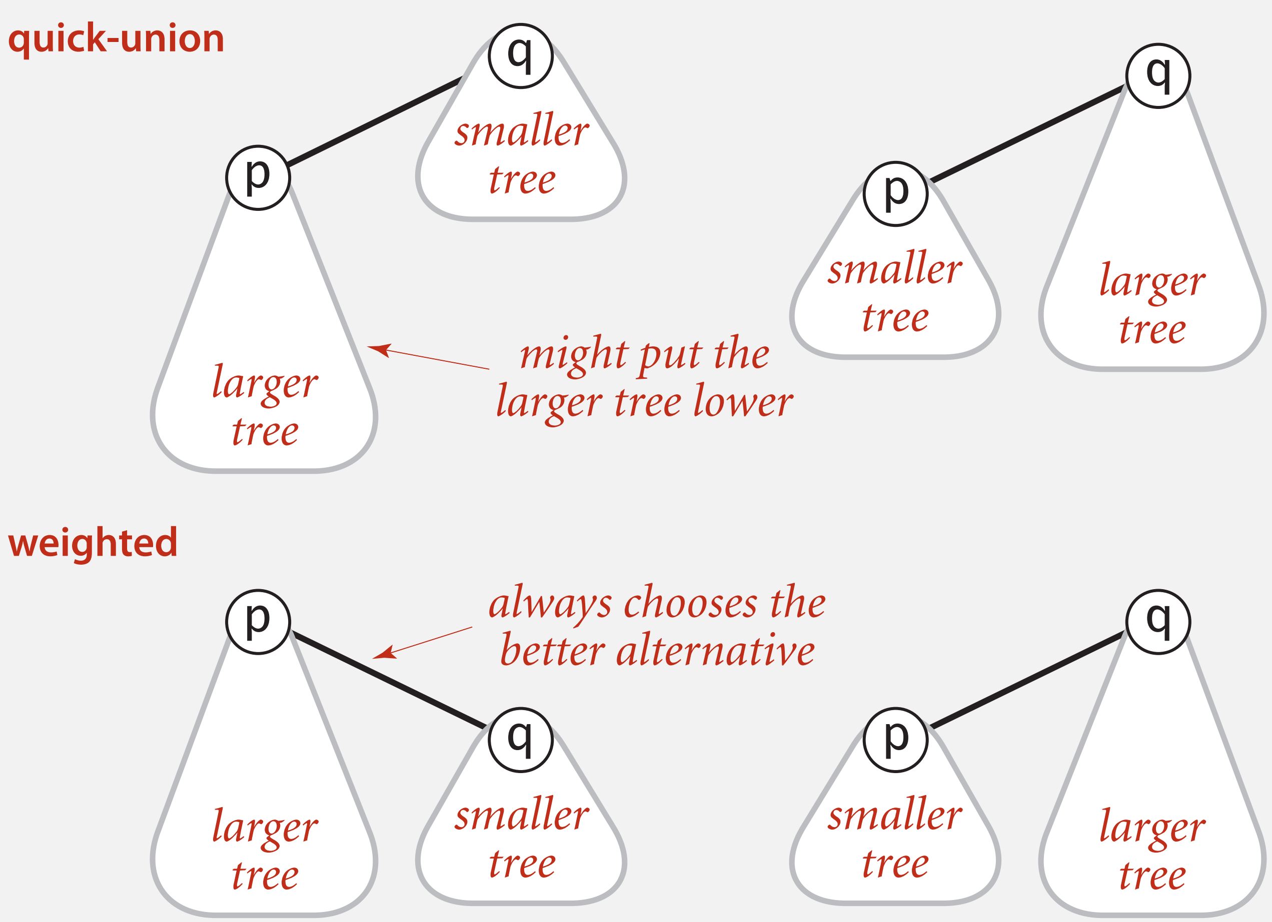
A. No, easy to improve further.

練習：想想看，如果利用高度(height)決定誰為 root，會比較好還是比較差。

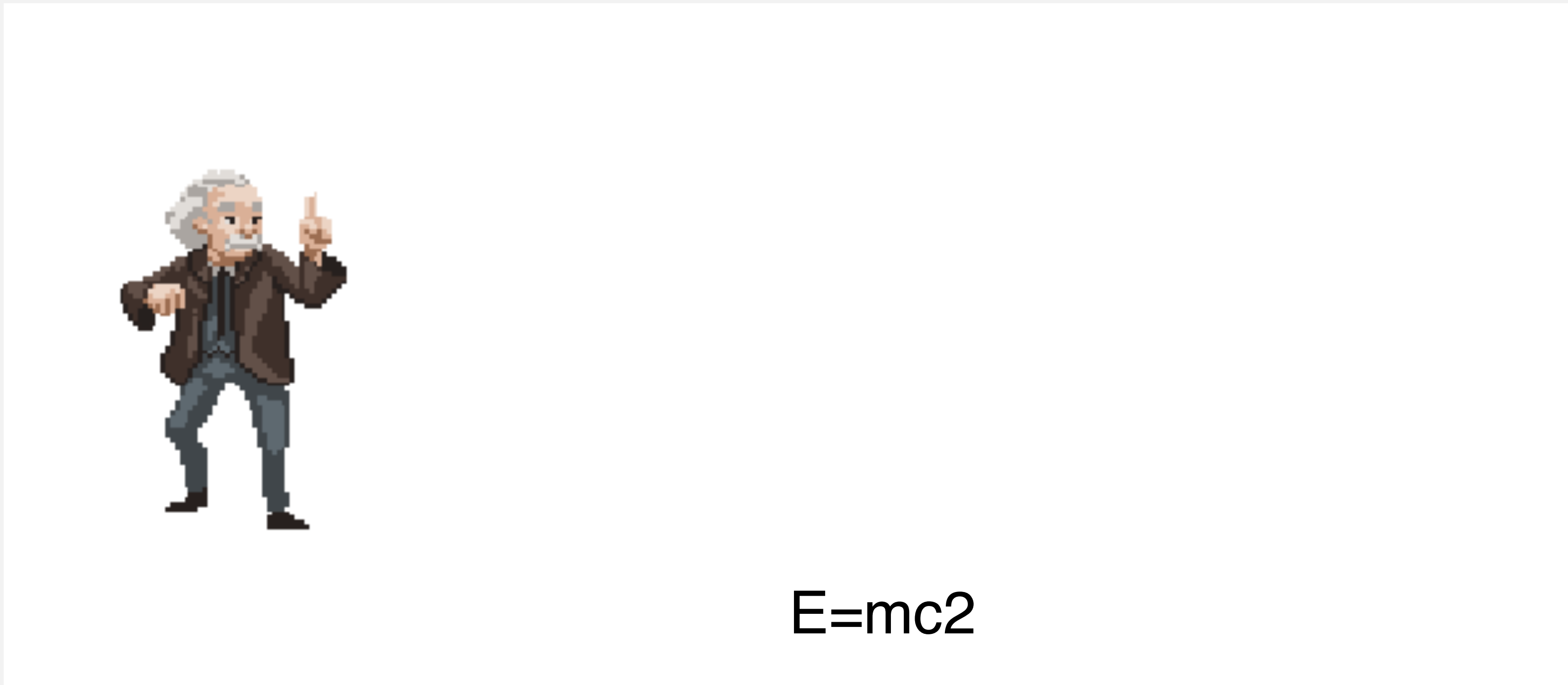
Weighted quick-union.

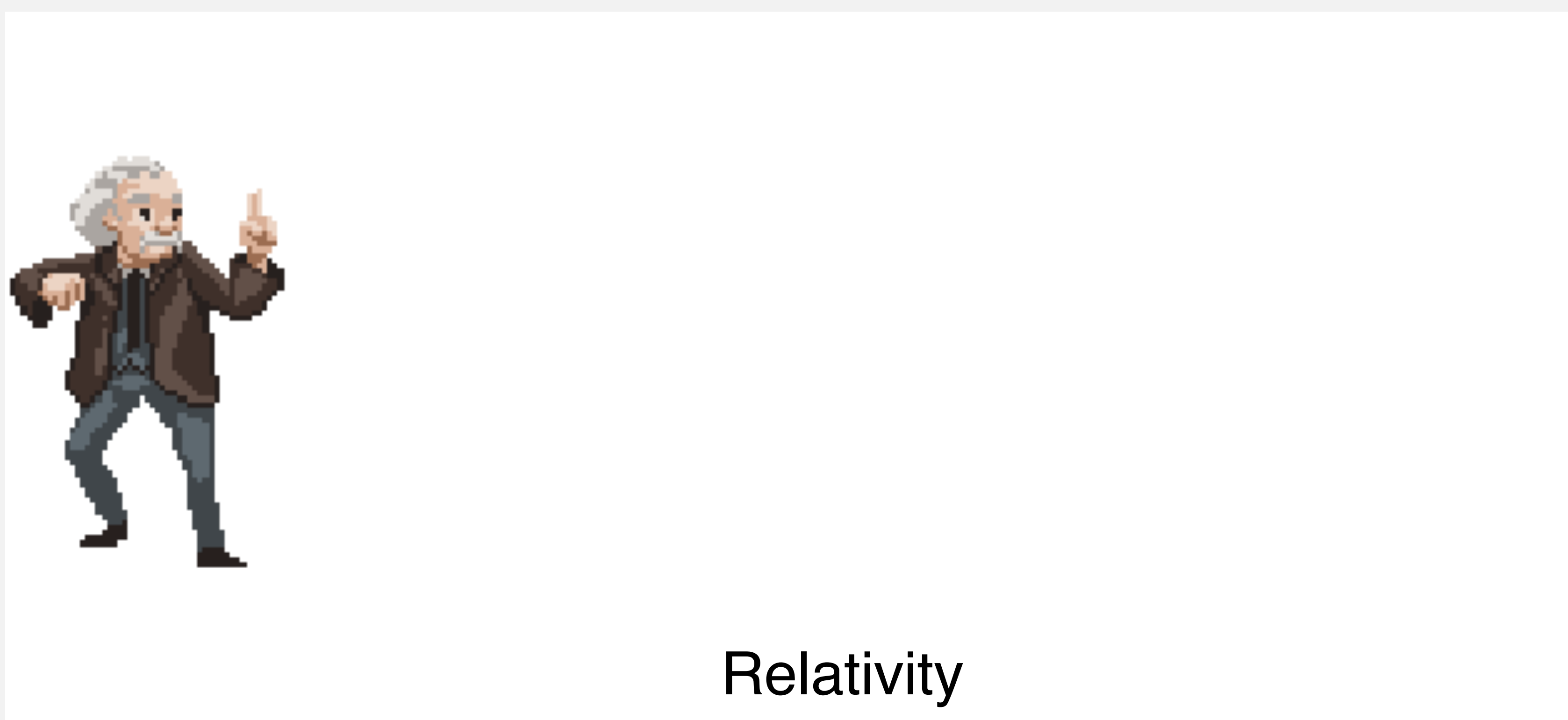
- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

reasonable alternatives:
union by height or "rank"



Rest





Any Other Possible Way to Speed Up ?

- Ideally, we would like every node to link directly to the root of its tree, but we do not want to pay the price of changing a large number of links, as we did in the quick-find algorithm.
- We can approach the idea simply by making all the nodes that we do examine directly link to the root.

only one extra line of code !

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

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

Path compression: Java implementation

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

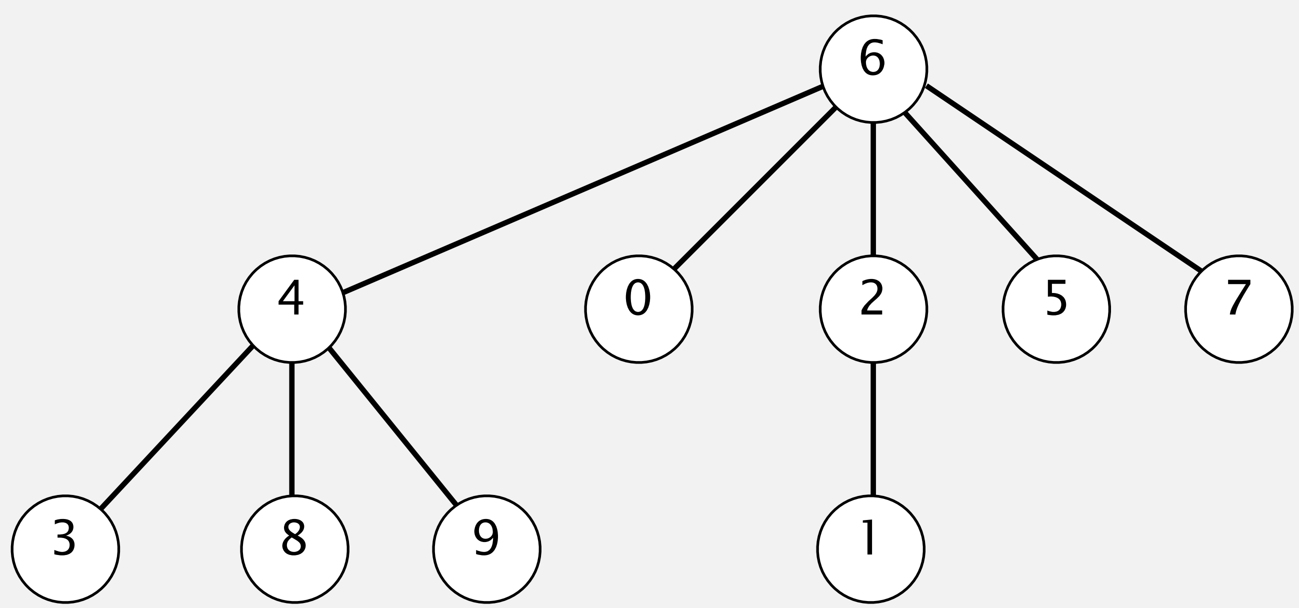
Simpler one-pass variant (path halving): Make every other node in path point to its grandparent.

```
public int find(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.

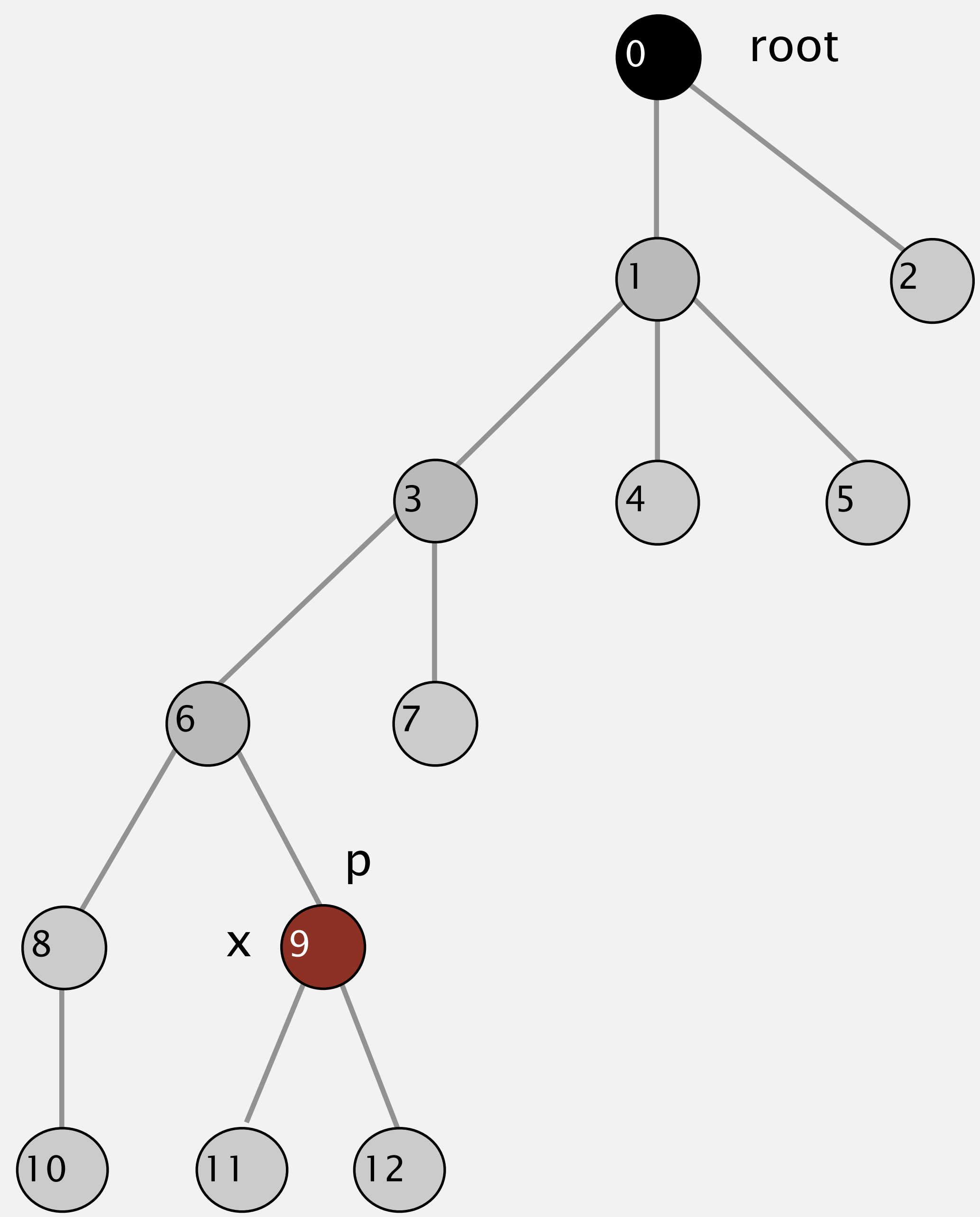
Weighted quick-union with path compression demo



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

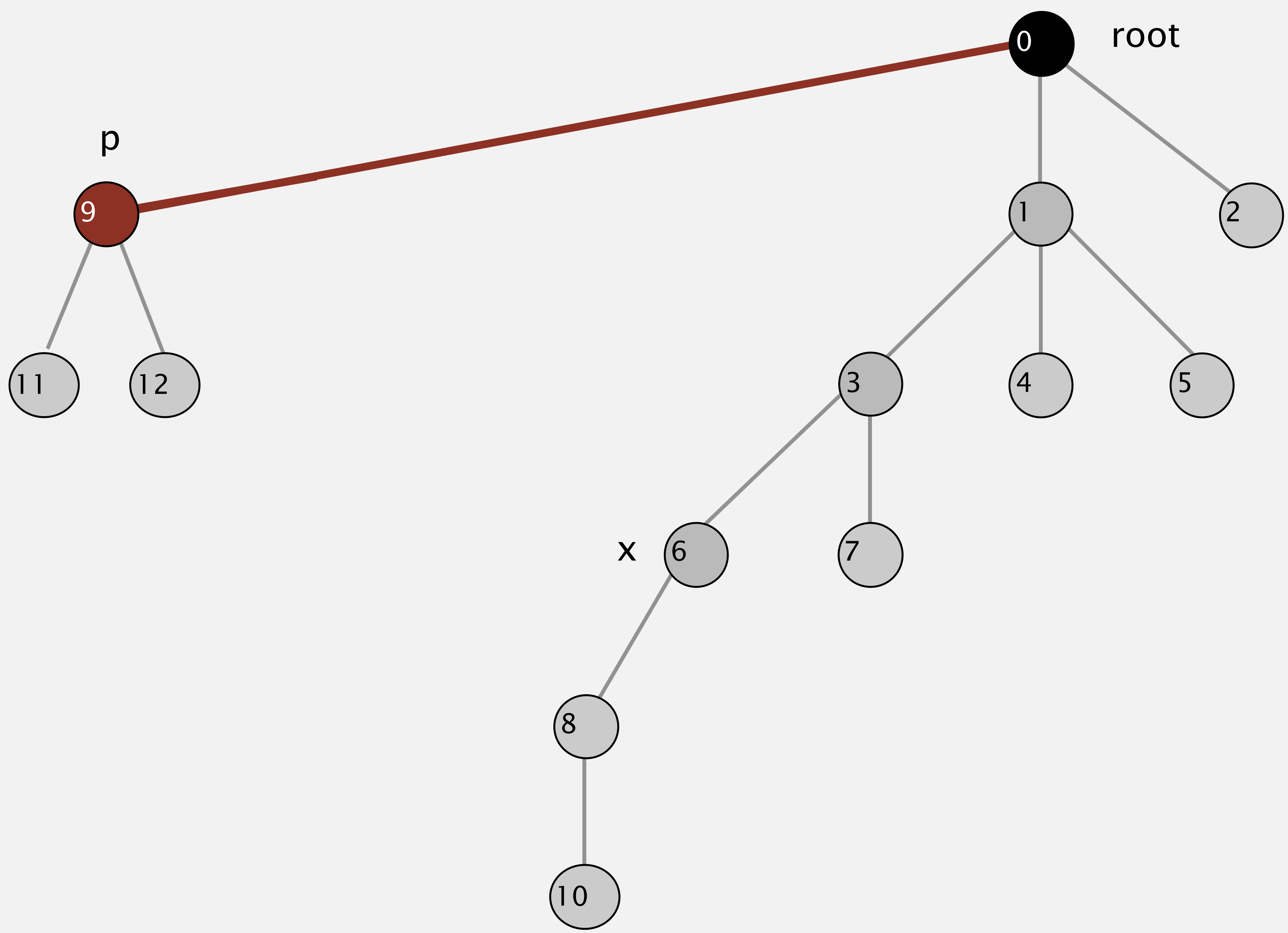
Improvement 2: path compression

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



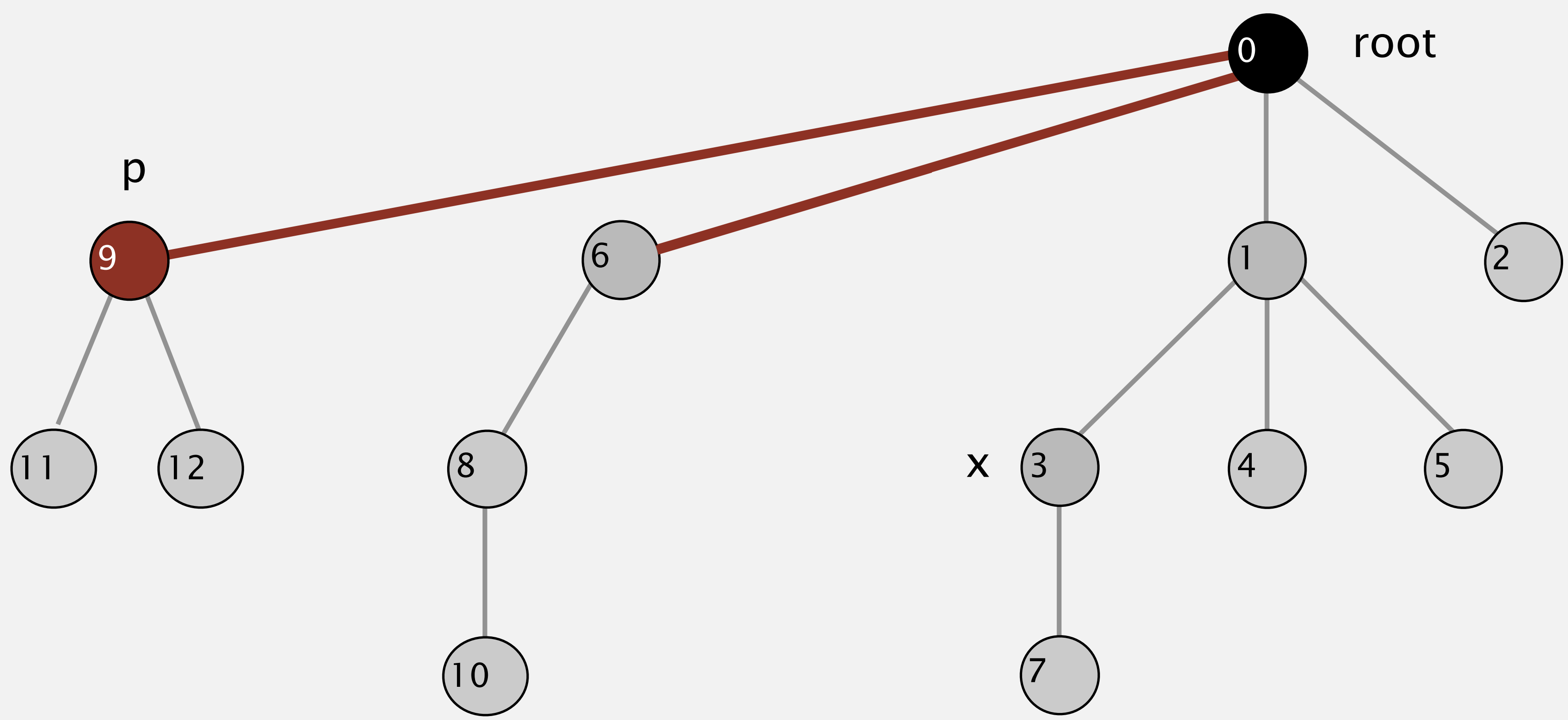
Improvement 2: path compression

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



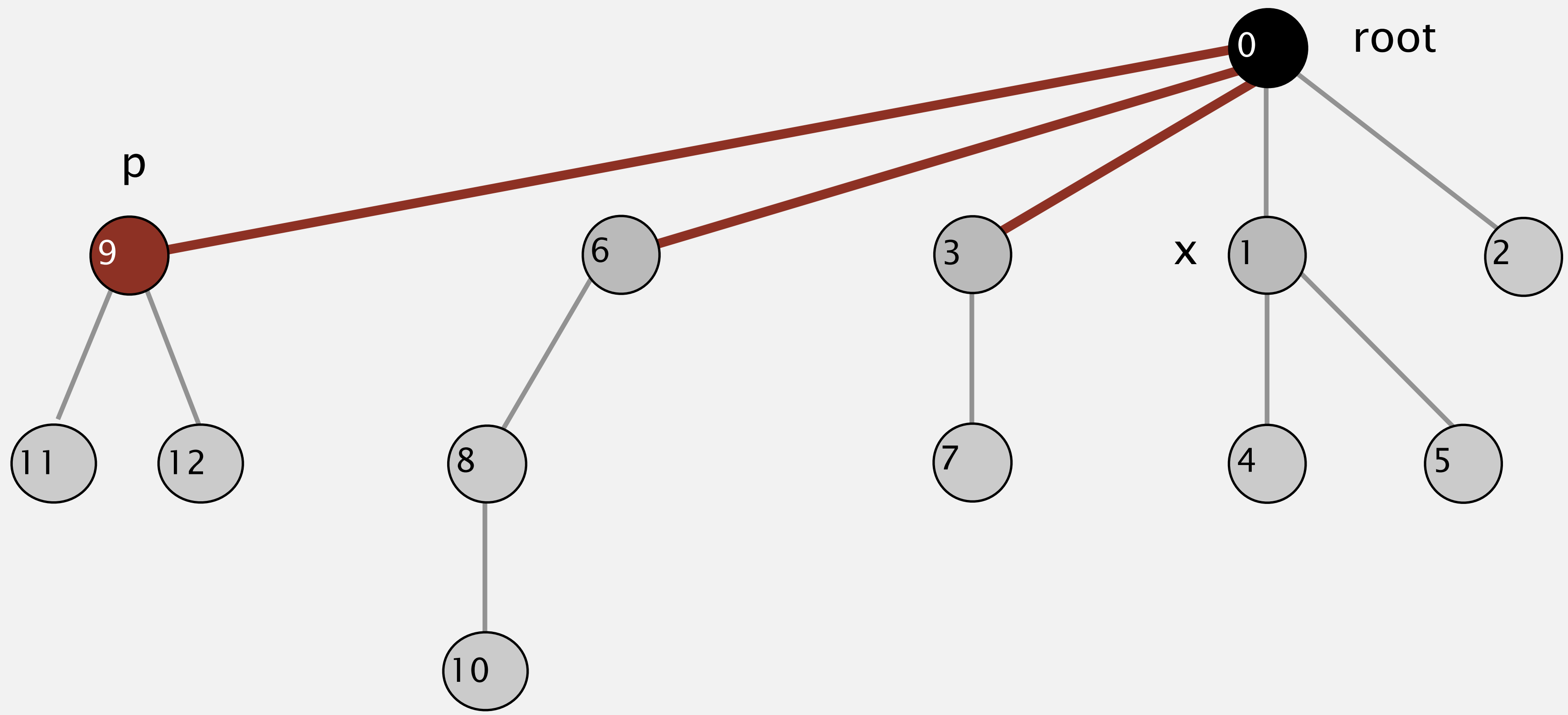
Improvement 2: path compression

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



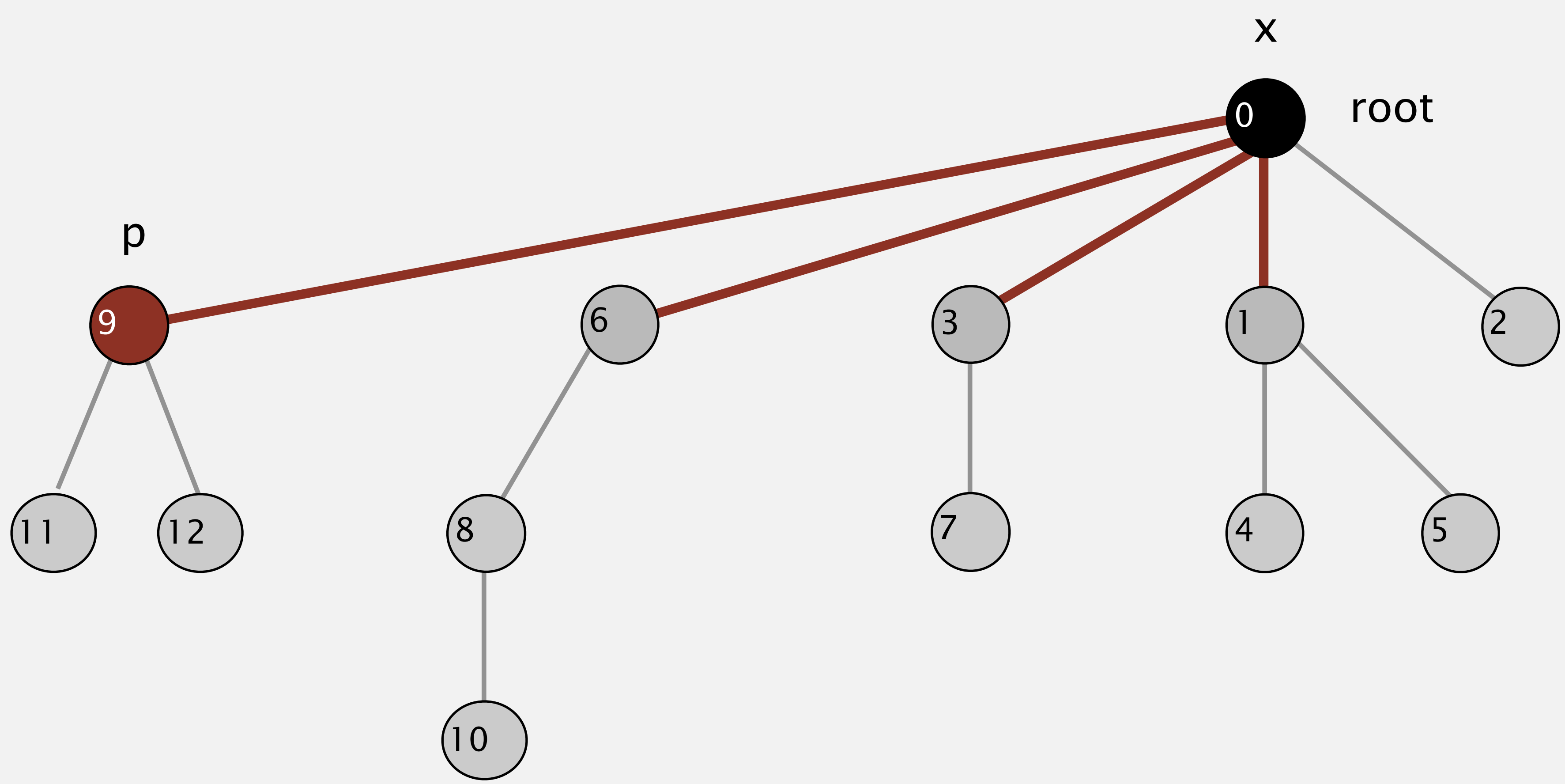
Improvement 2: path compression

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

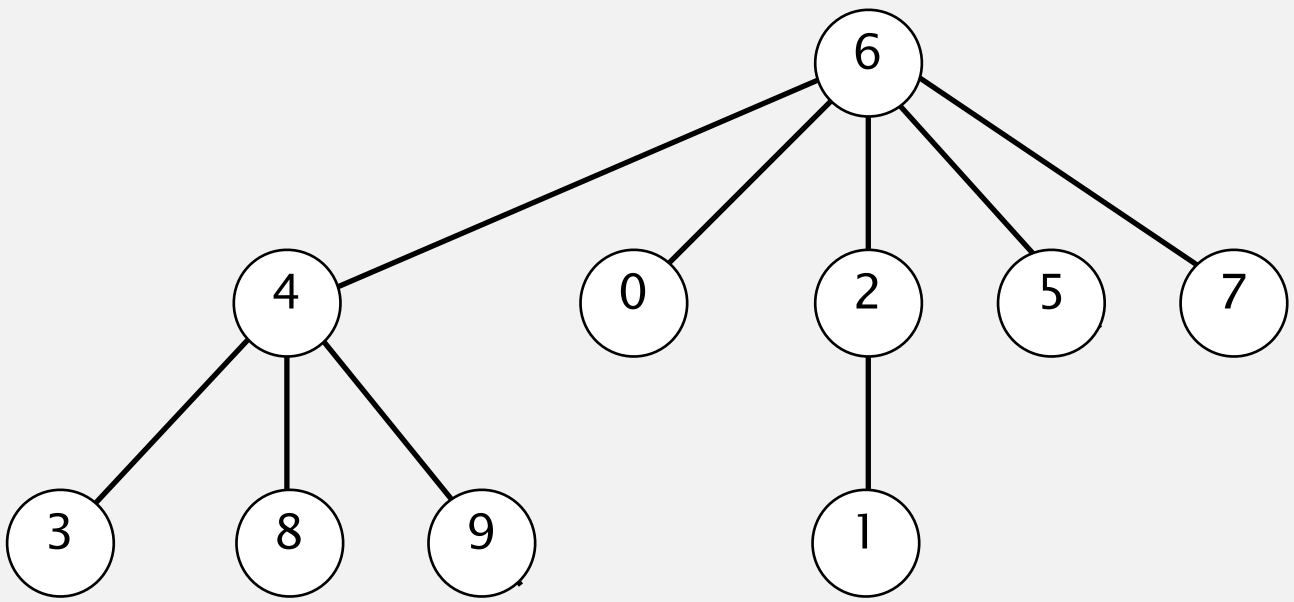
Quick union with path compression. Just after computing the root of p, set the id[] of each examined node to point to that root.



Bottom line. Now, find() has the side effect of compressing the tree.

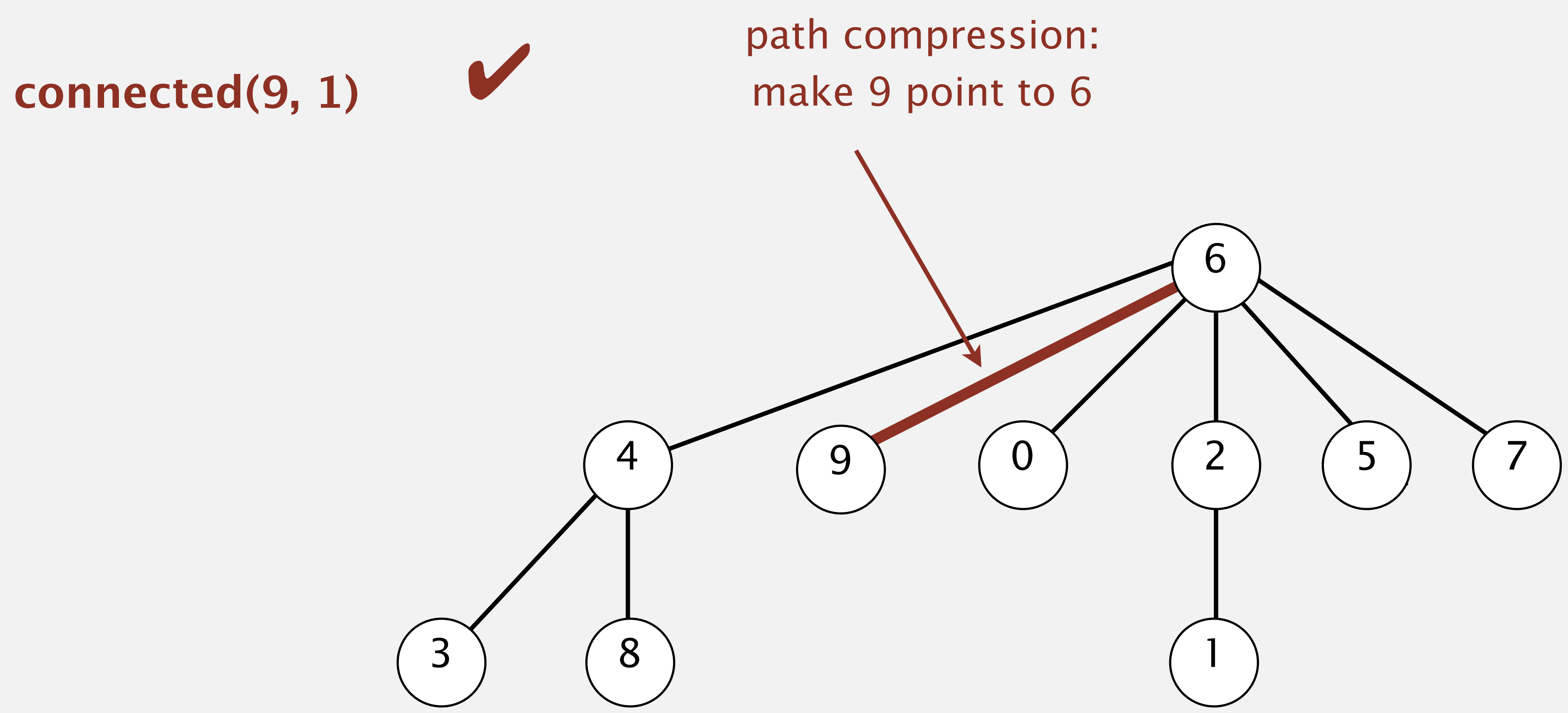
Weighted quick-union with path compression demo

connected(9, 1) ✔



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

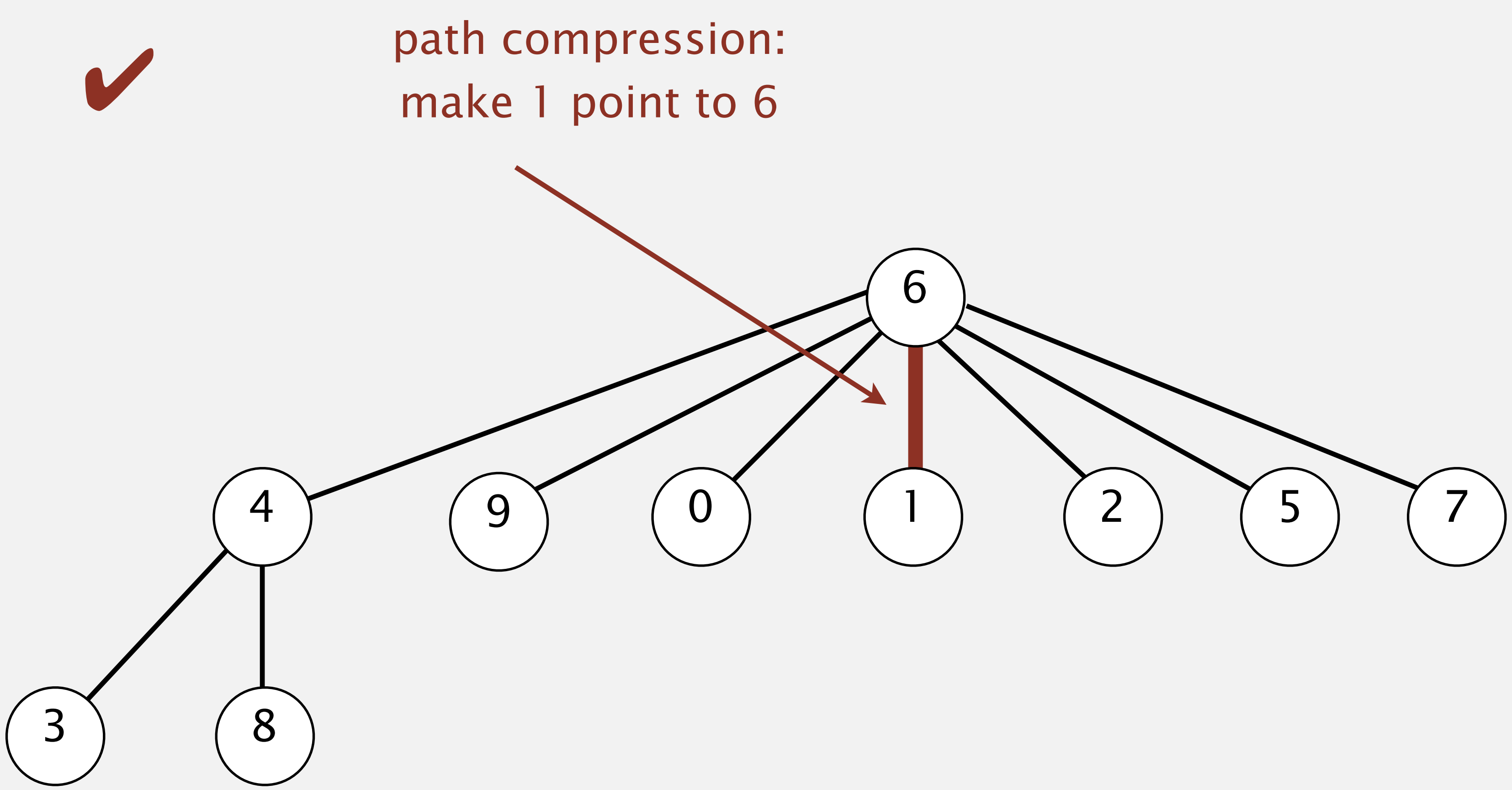
Weighted quick-union with path compression demo



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

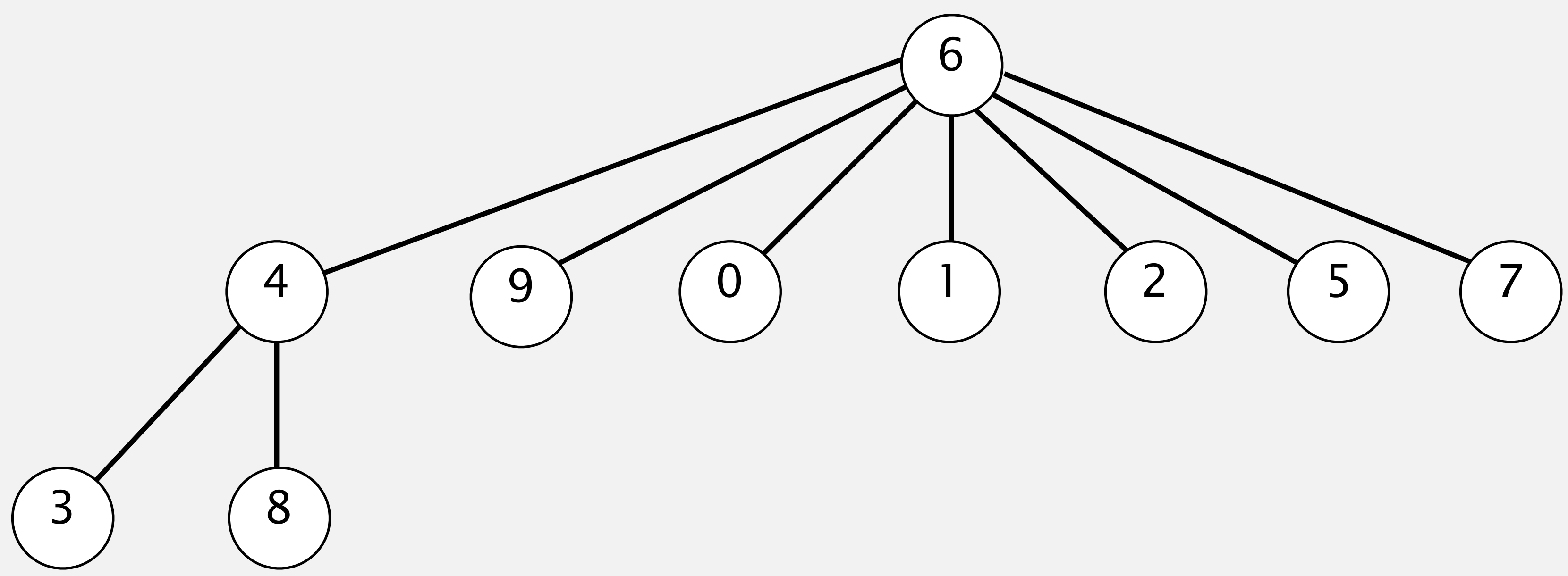
Weighted quick-union with path compression demo

connected(9, 1) ✔



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

Weighted quick-union with path compression demo



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

Summary

Key point. Weighted quick union (and/or 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

order of growth for M union-find operations on a set of N objects

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

iterated lg function

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.

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.