CSCI2100B Data Structures
Union-Find

Irwin King

king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Outline

- Dynamic Connectivity
- Quick Find
- Quick Union
- Improvements
- Applications

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

Purpose

- Learning the 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

Outline

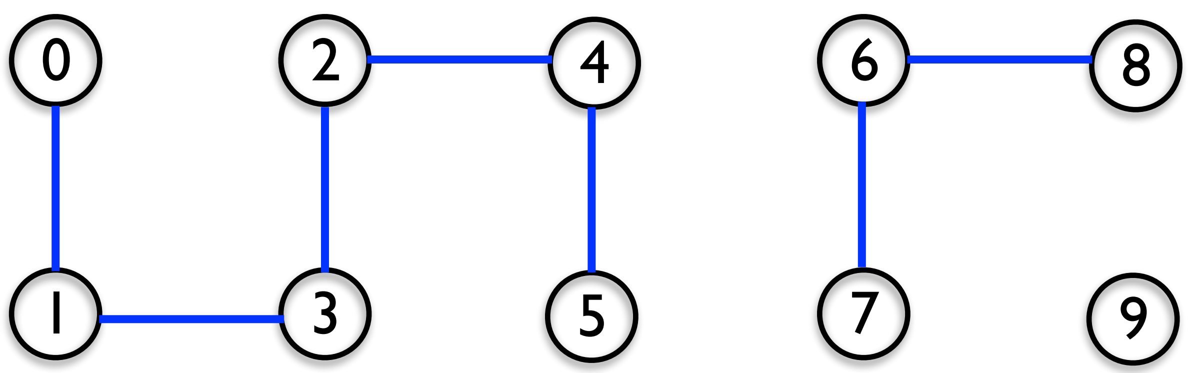
- Dynamic Connectivity
- Quick Find
- Quick Union
- Improvements
- Applications

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

Dynamic Connectivity

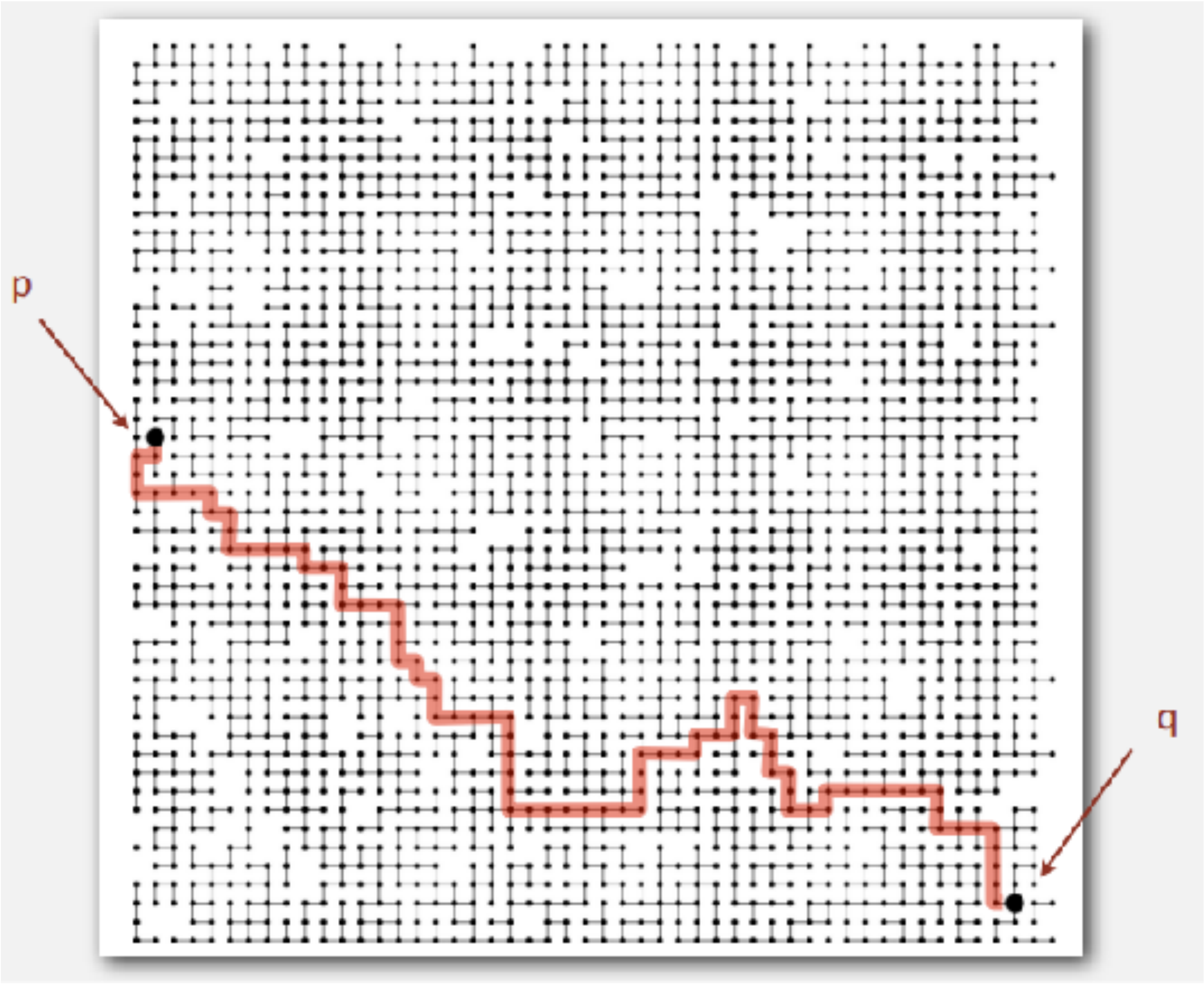
- Given a set of *N* objects.
  - Union Command: connect two objects
  - Find/connected query: is there a path connecting the two objects?

union(6, 8)
union(1, 3)
union(2, 4)
union(2, 3)
union(6, 7)
connected(0, 5) NO
connected(7, 8) YES
union(0, 1)
union(4, 5)
connected(0, 5) YES



Example

- Q. Is there a path connecting p to q ?



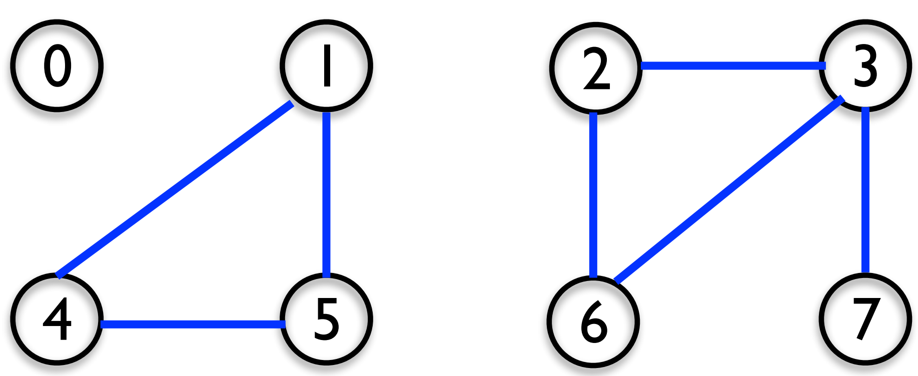
- A. Yes

Modeling the Objects

- Applications involve manipulating objects of all types.
  - Pixels in a digital photo
  - Computers in a network
  - Friends in a social network
  - Elements in a mathematical set
- Naming objects 0 to N-1 is convenient when programming
  - Use integers as array index
  - Suppress details not relevant to union-find

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

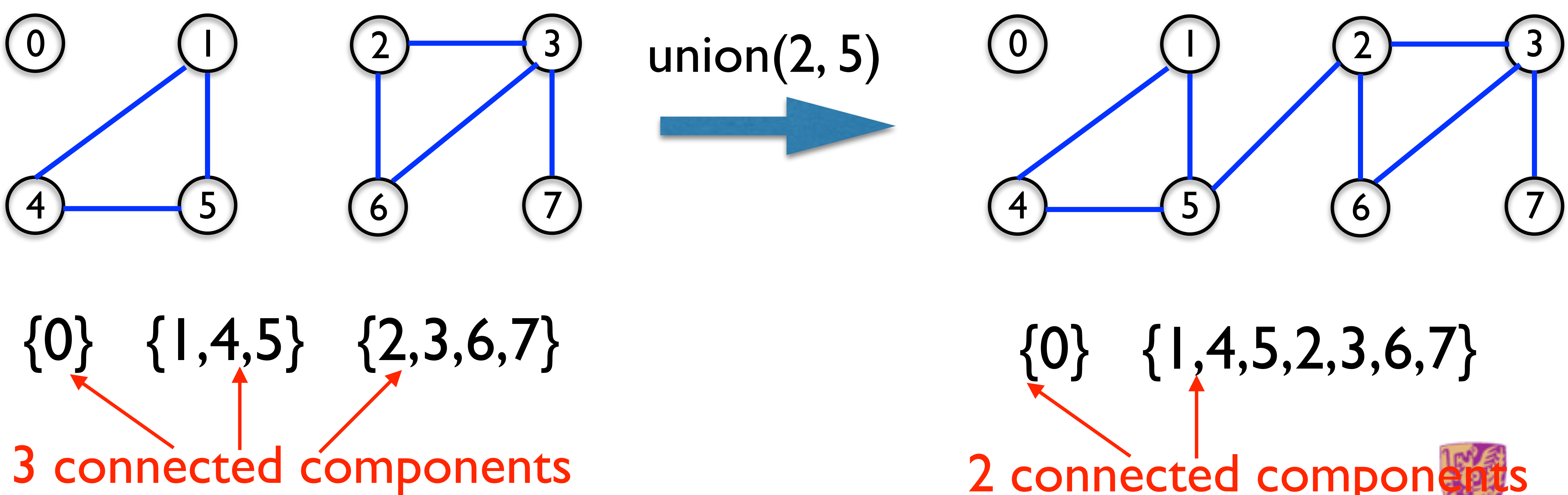


{0} {1,4,5} {2,3,6,7}

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 Structure

- 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

Outline

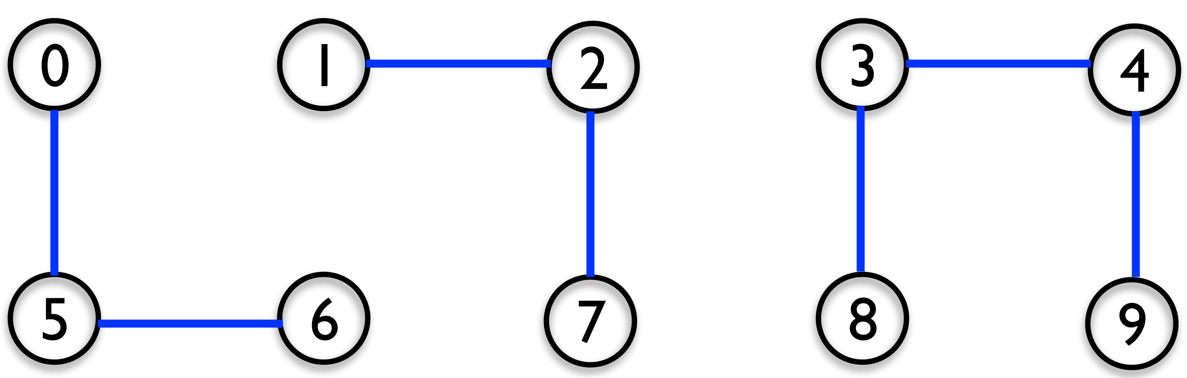
- Union-Find Problem
- Quick Find
- Quick Union
- Improvements
- Applications

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

Quick-find

- Data structure
  - Integer array id[] of length *N* (the number of objects)
  - Interpretation: *p* and *q* are connected iff (if and only if) they have the same id

	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

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

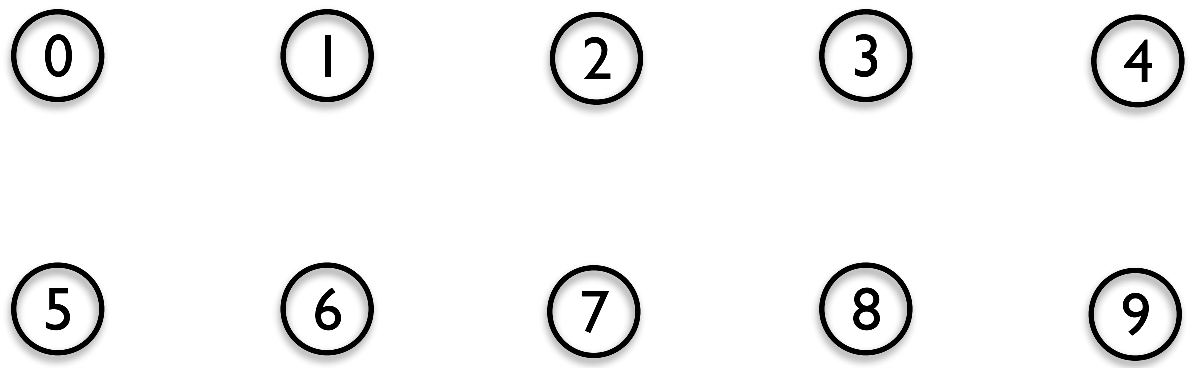
- Find: Check is p and q have the same id
  - id[6] == 0; id[1] == 1; 6 and 1 are not connected
- Union: To merge components containing p and q, change all entires whose id equals id[p] to id[q]

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

p q
after union(6, 1)

Problem: many values can change

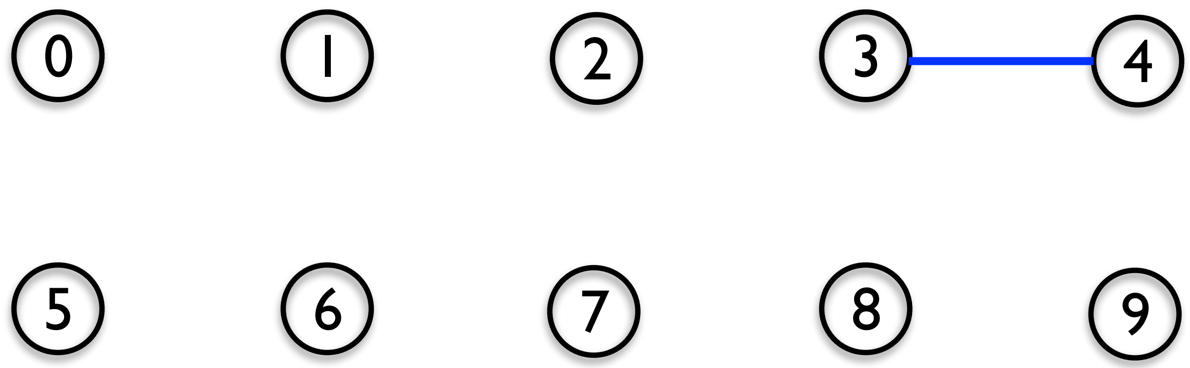
Quick-find Demo



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

Initial state: no any connection, id[i] == i.

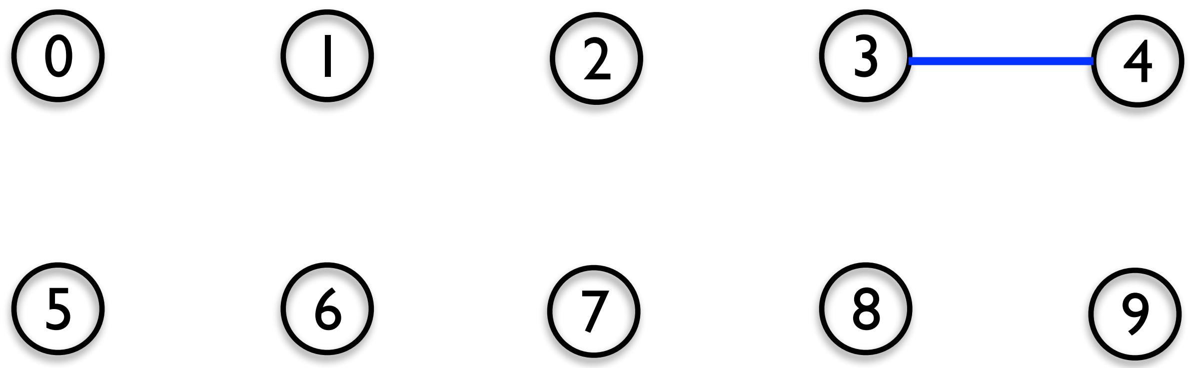
Quick-find Demo



union(4, 3)

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

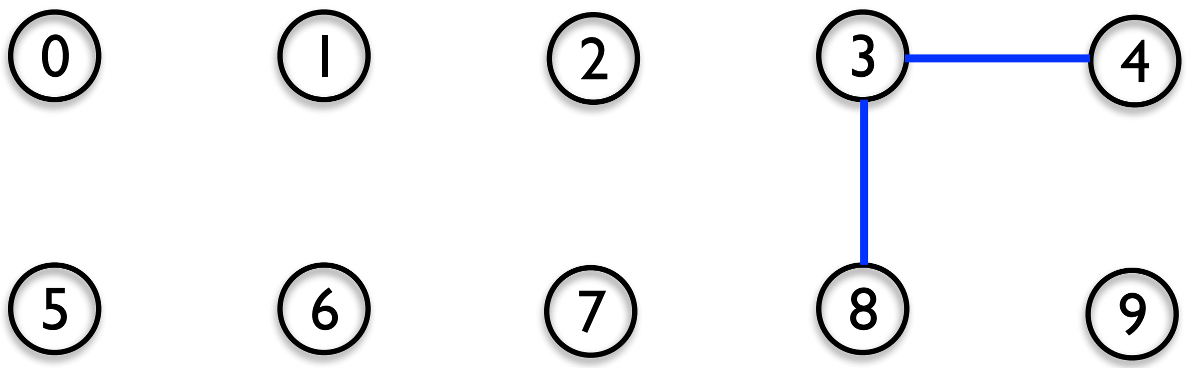
Quick-find Demo



union(4, 3)

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

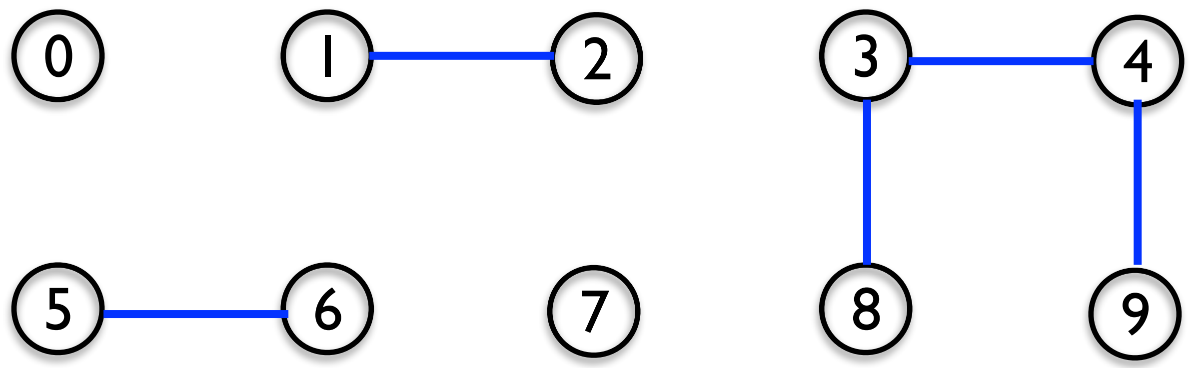
Quick-find Demo



union(3, 8)

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

Quick-find Demo



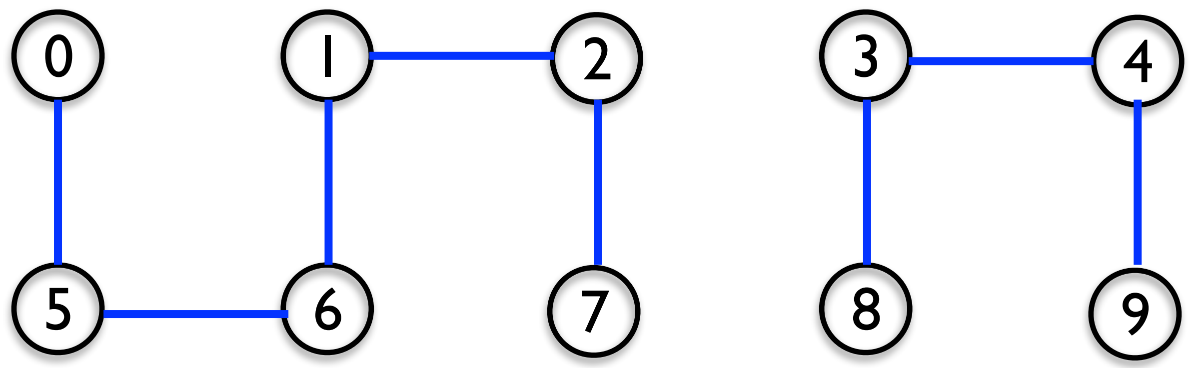
union(6, 5)
union(9, 4)
union(2, 1)
connected(8, 9) YES

Since id[8]==id[9]

connected(5, 0) NO

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

Quick-find Demo



union(5, 0)
union(7, 2)
union(6, 1)
connected(5, 0) YES

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

Quick-find Implementation

```
void QuickFindIni(int * id[], int N)
{
      int i;
      for(i = 0; i < N; i++)
            id[i] = i;
}
```

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

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

Check whether *p* and *q* are in the same component
(2 array accesses)

```
void union(…, int p, int q)
{
      int pid = id[p];
      int qid = id[q];
      int i;
      for(i = 0; i < N; 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

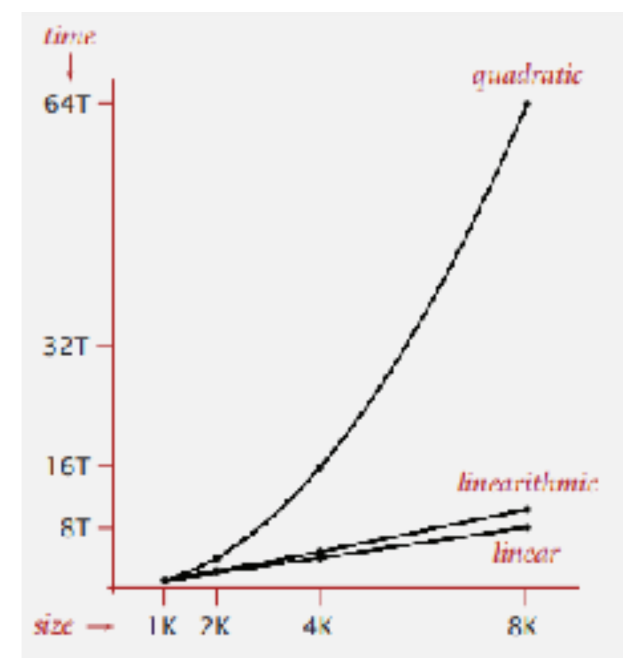
- Cost model: Number of array access (for read or write)

order of growth of number of array accesses

Algorithm	Initialize	Union	Find
quick-find	N	N	1

- Quick-Find defect: Union too expensive
  - Ex. Takes N^2 array accesses to process of N union commands on N objects.

quadratic



Outline

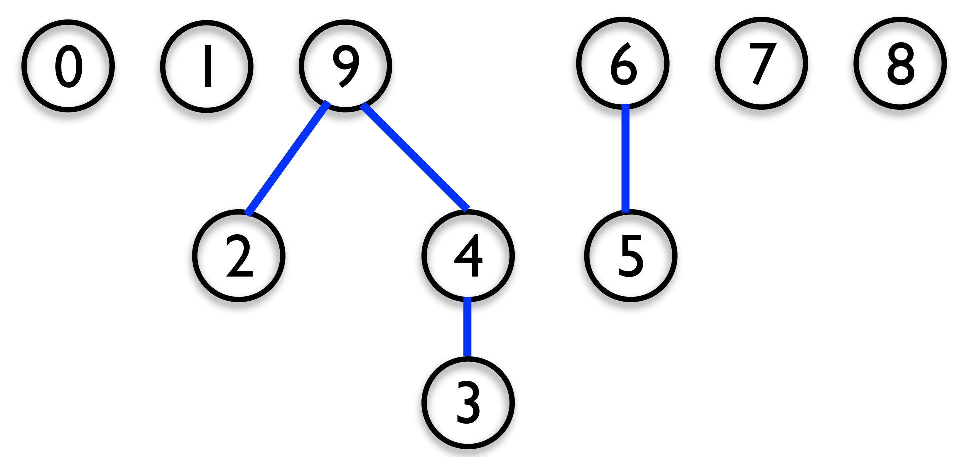
- Dynamic Connectivity
- Quick Find
- Quick Union
- Improvements
- Applications

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

Quick-union

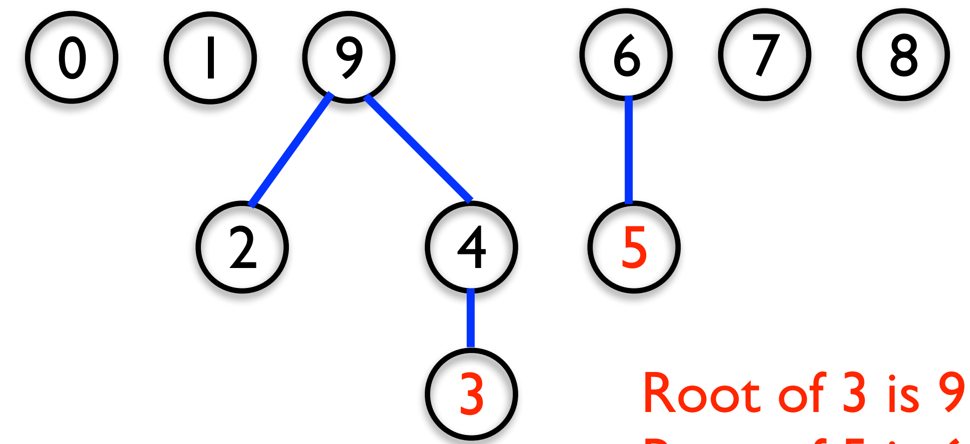
- Data structure
  - Integer array id[] of length *N* (the number of objects)
  - Interpretation: id[i] is parent of i
  - Root of i is id[id[id[…id[i]…]]]. Keep going until it doesn't change (algorithm ensures no cycles)

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



Quick-union

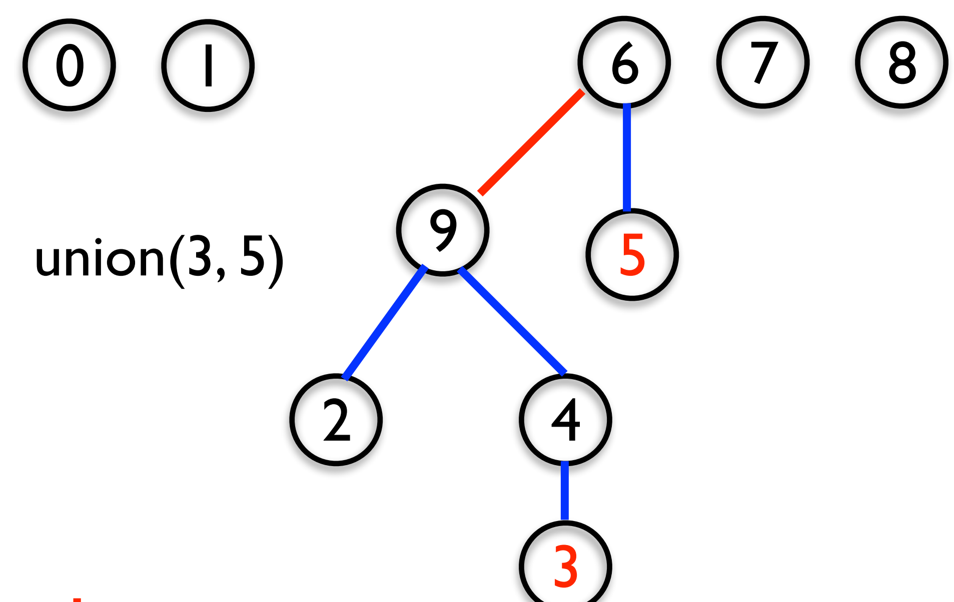
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



Root of 3 is 9
Root of 5 is 6
3 and 5 are not connecte

- **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

union(3, 5)



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

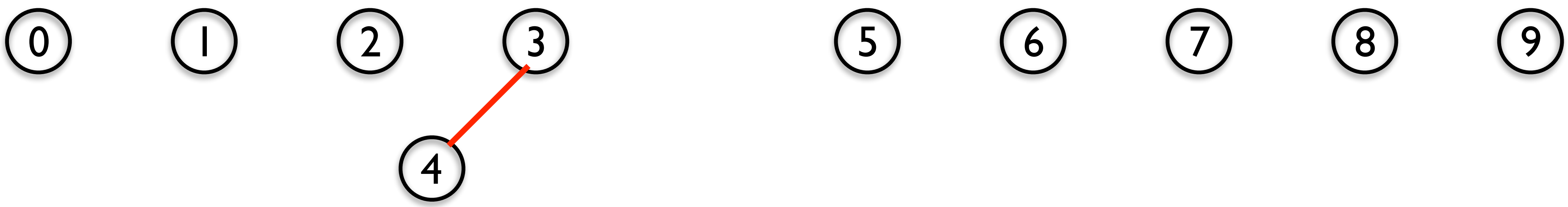
only one value changes

Quick-union 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

union(4, 3)



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

Quick-union Demo

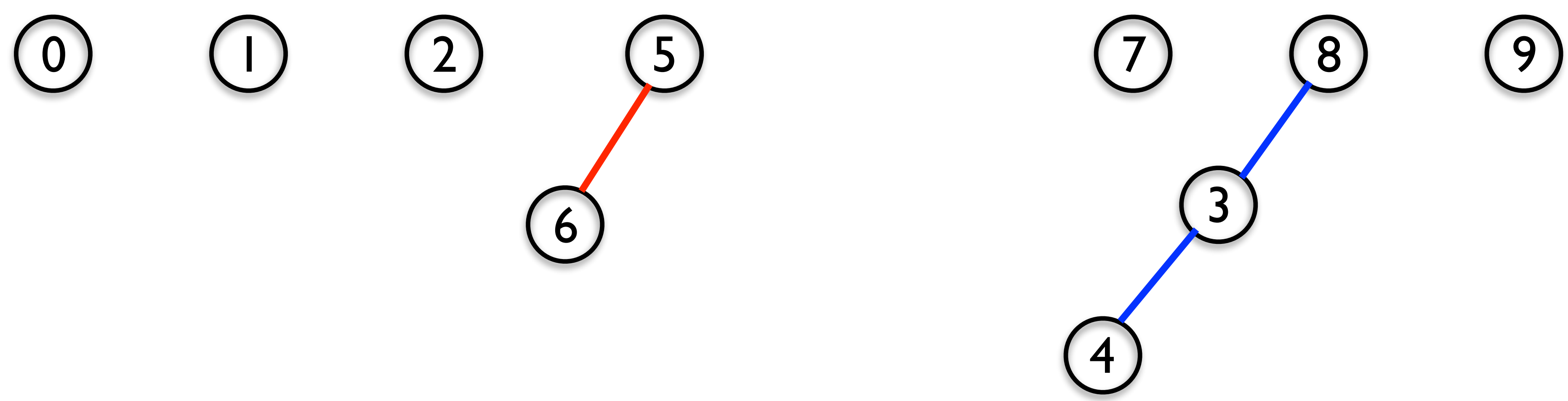
union(3, 8)

id[]

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

Quick-union Demo

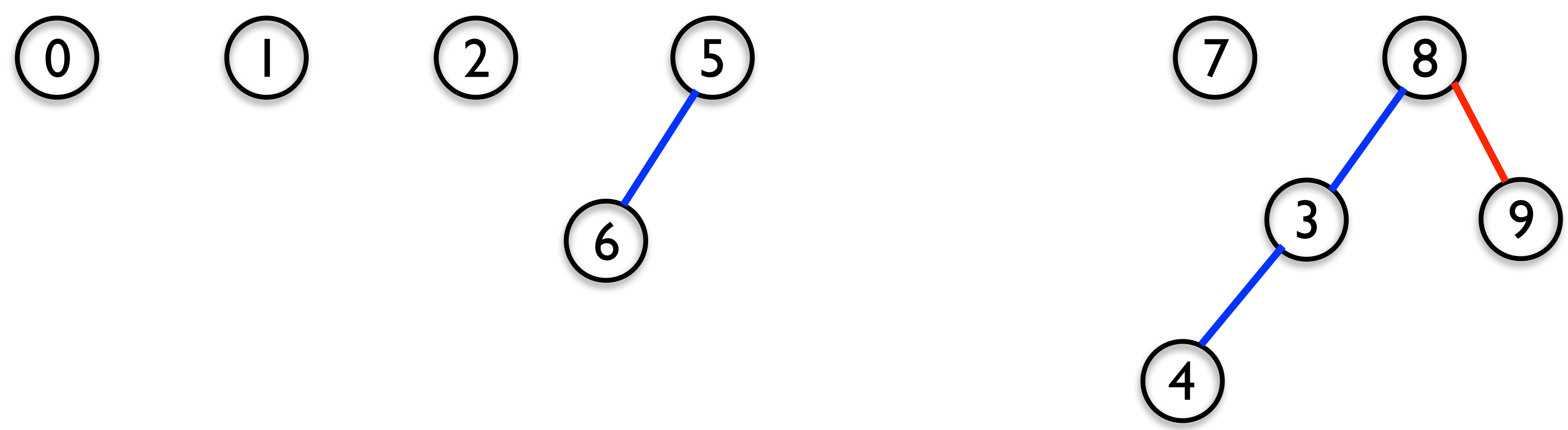
union(6, 5)



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

Quick-union Demo

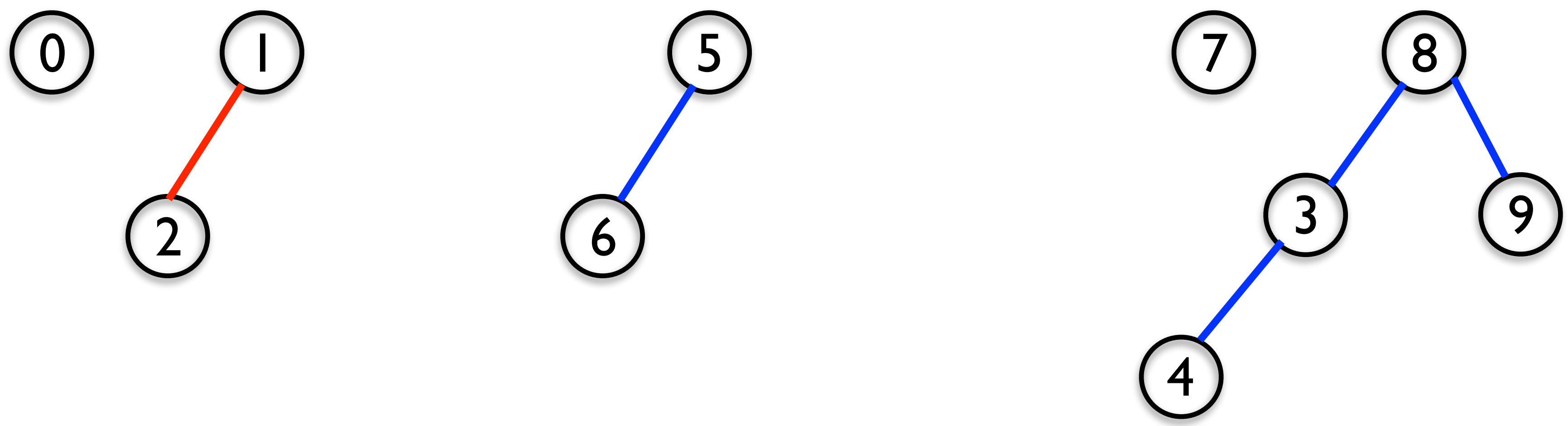
union(9, 4)



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

Quick-union Demo

union(2, 1)

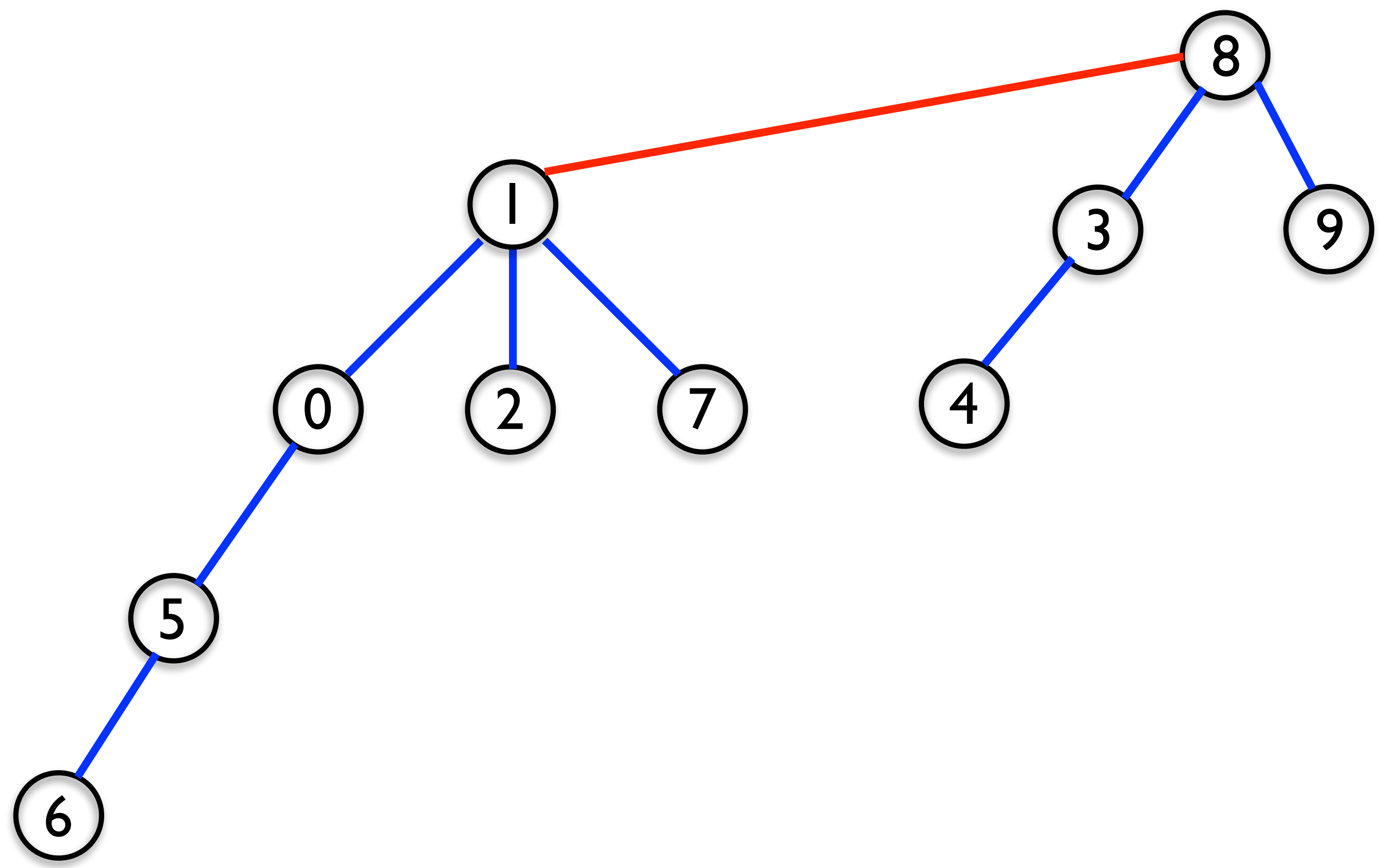


id[]

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

Quick-union Demo

union(5, 0)
union(7, 2)
union(6, 1)
union(7, 3)



id[]

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

Quick-union Implementation

```
void QuickUnionIni(int * id[], int N)
{
        int i;
        for(i = 0; i < N; i++)
                id[i] = i;
}
```

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

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

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

Quick-union Implementation

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

Check whether *p* and *q* have same root (depth of *p* and *q* array accessed)

```
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 access (for read or write)

order of growth of number of array accesses

Algorithm	Initialize	Union	Find
quick-find	*N*	*N*	1
quick-union	*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 then flat
- Quick-union defect
  - Trees can get very tall
  - Find the root is too expensive (could be *N* array accesses)

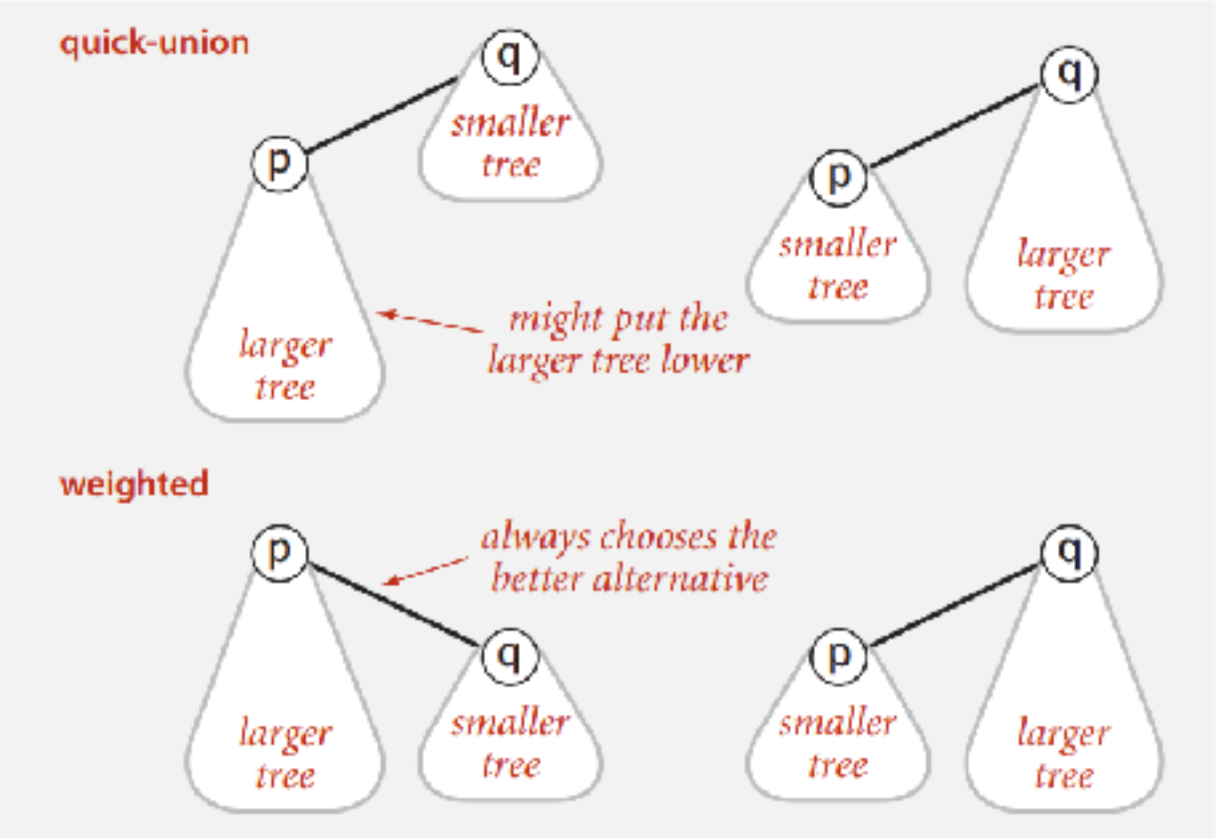
Outline

- Dynamic Connectivity
- Quick Find
- Quick Union
- Improvements
- Applications

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

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



reasonable alternatives: union by height

Weighted Quick-union Demo

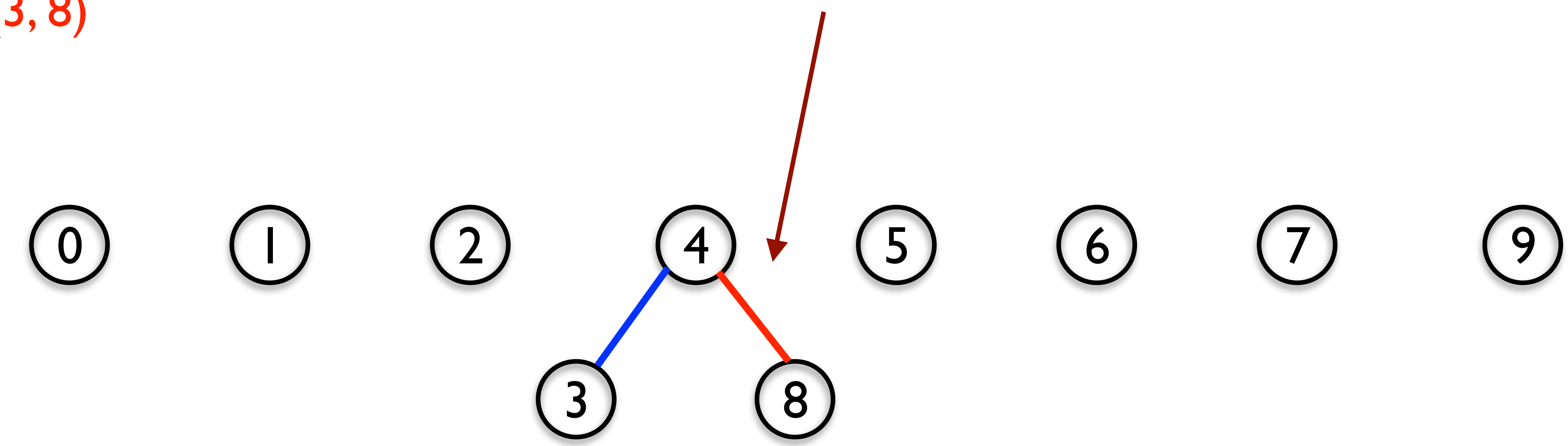
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

Weighted Quick-union Demo

union(4, 3)

union(3, 8)

weighting: make 8 point to 4 (instead of 4 to 8)

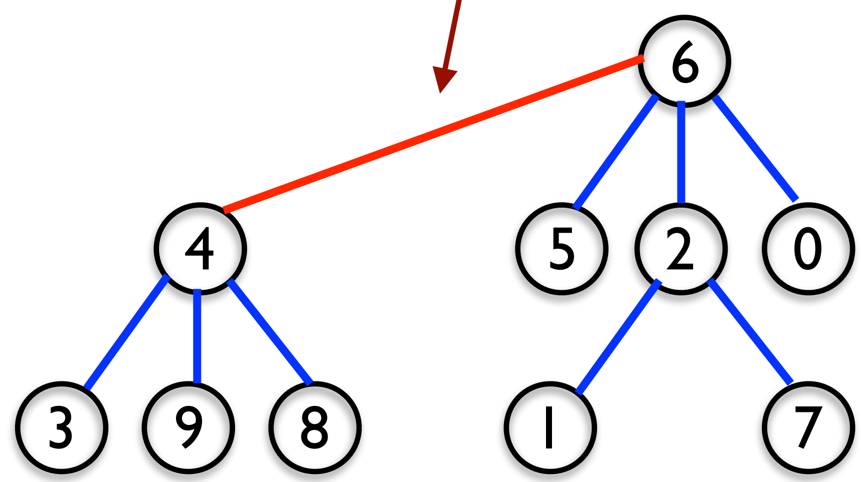


	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	4	4	5	6	7	4	9

Weighted Quick-union Demo

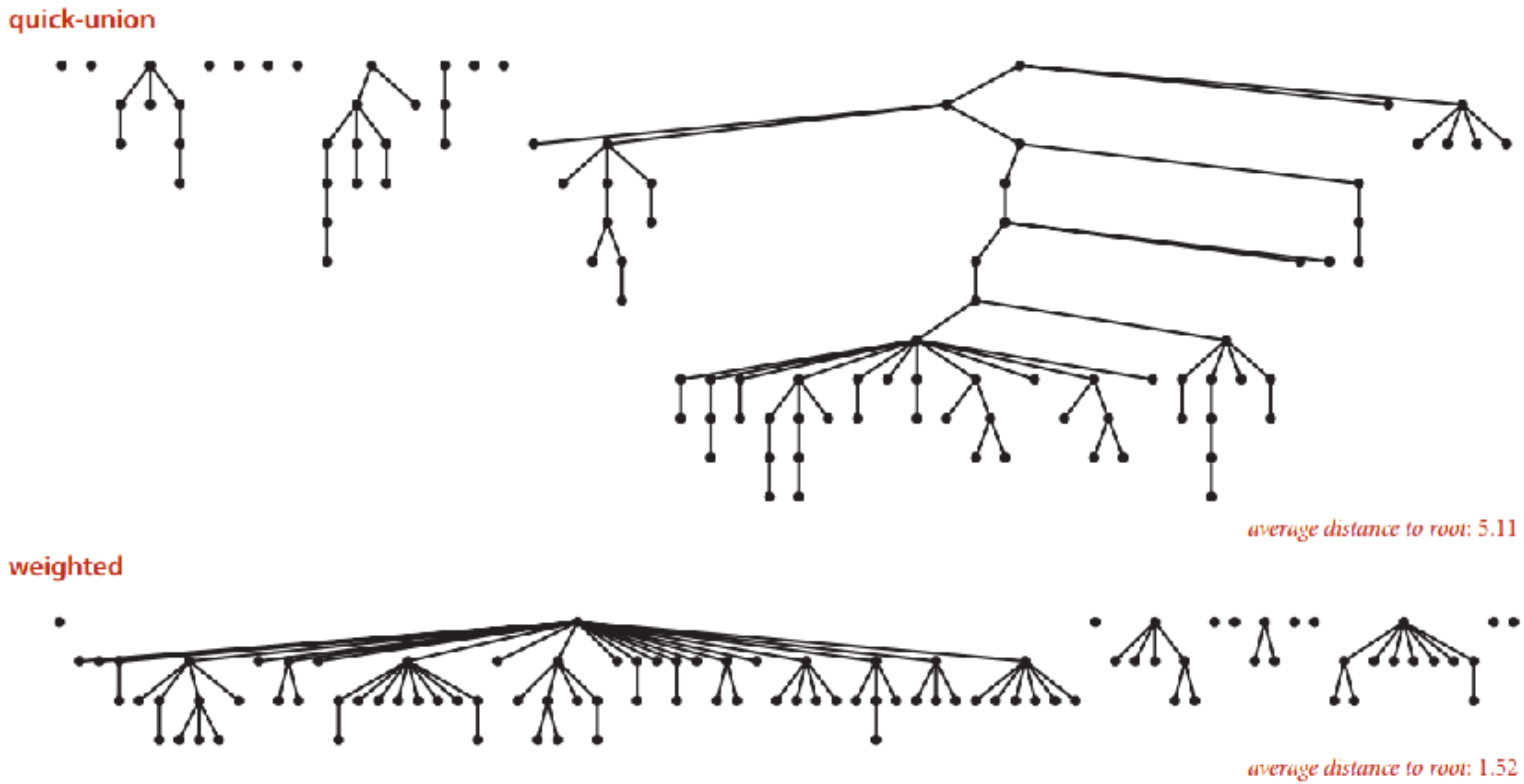
union(6, 5)
union(9, 4)
union(2, 1)
union(5, 0)
union(7, 2)
union(6, 1)
union(7, 3)

weighting: make 4 point to 6 (instead of 6 to 4)



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

Example



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

Weighted Implementation

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

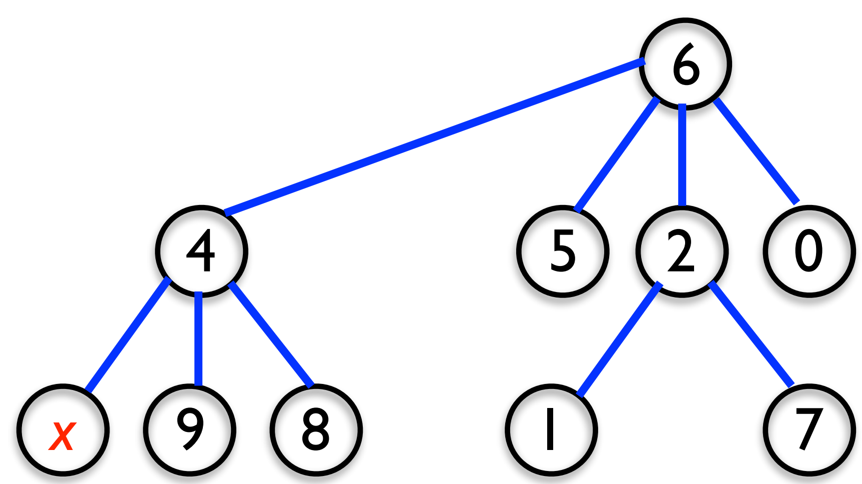
- Running time
  - Find (mainly for getting roots): takes time proportional to depth of p and q
  - Union: takes constant time, given roots

lg = base-2 logarithm

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

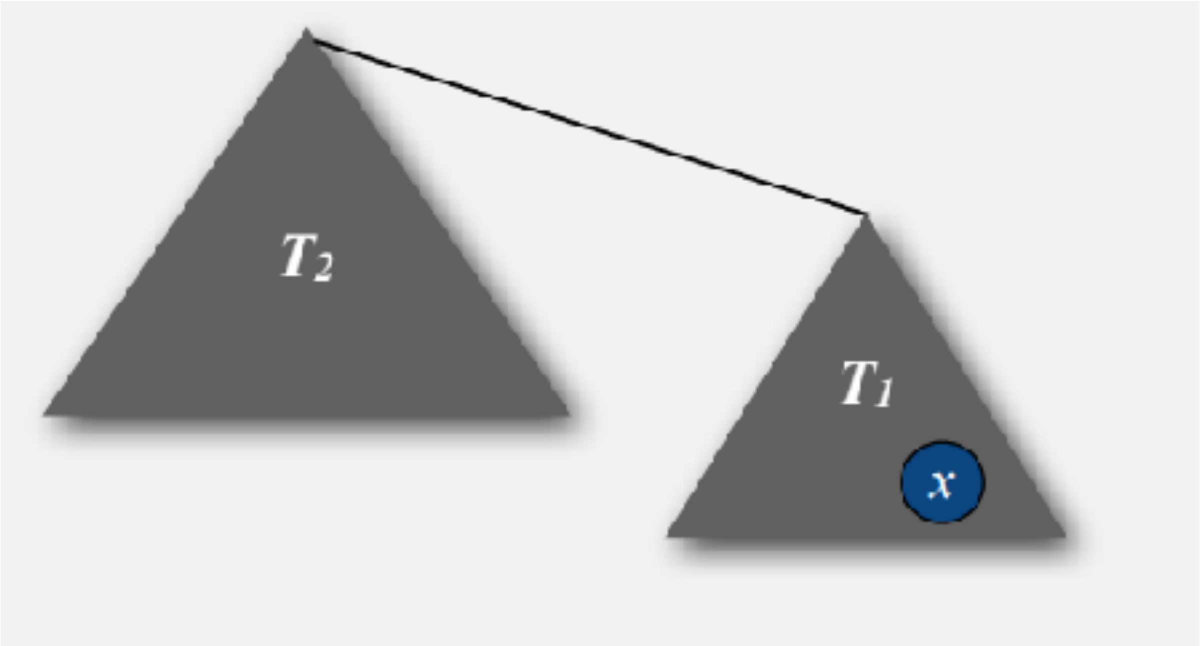
$N = 10$

Depth(x) $= 2 <= \lg N$



Weighted Quick-union Analysis

- Proposition: Depth of any node *x* is at most *lg N*
- Pf. When does depth of *x* increase ?
    - Increases by 1 when tree *T1* containing *x* is merged into another tree *T2*
        - The size of the tree containing x at least doubles since |*T2*| >= |*T1*|
        - Size of tree containing *x* can double at most *lg N* times



Weighted Quick-union Analysis

- Running time
  - Find (mainly for getting roots): takes time proportional to depth of p and q
  - Union: takes constant time, given roots
- Proposition: Depth of any node x is at most $lg\ N$

order of growth of number of array accesses

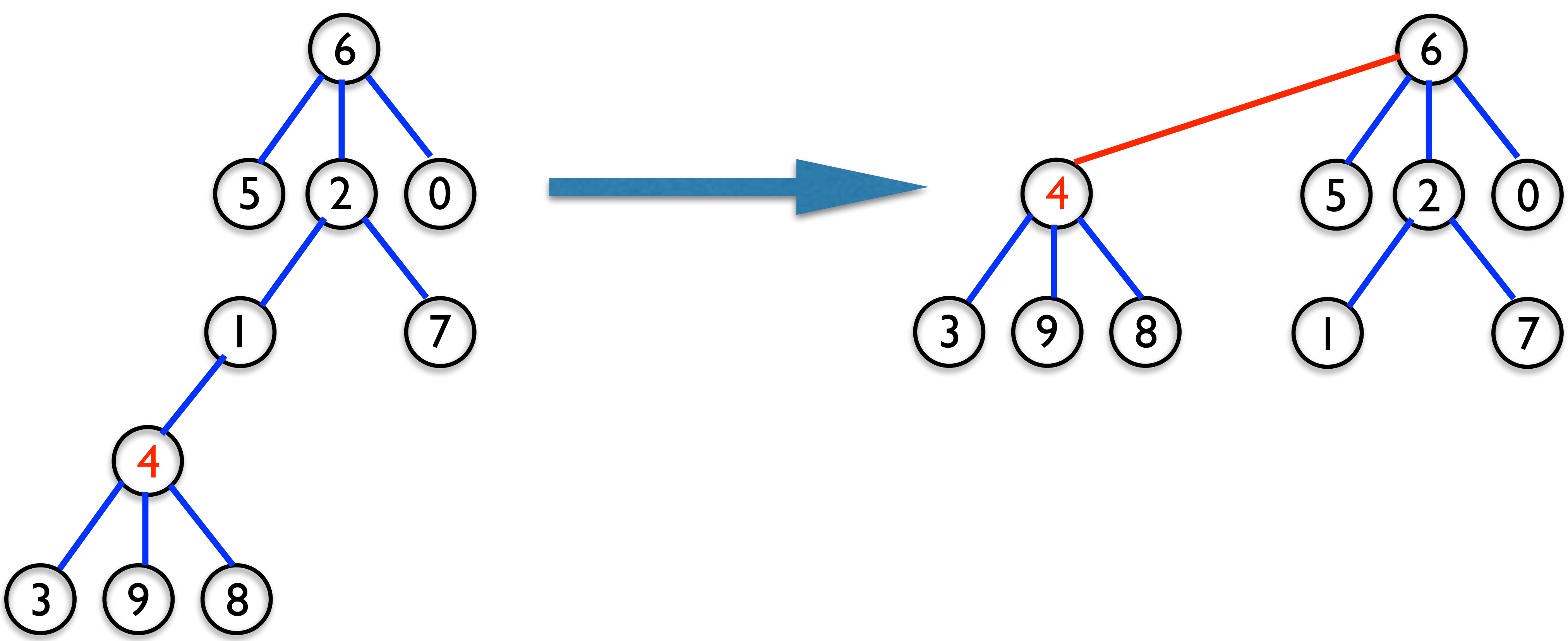
Algorithm	Initialize	Union	Find
quick-find	N	N	1
quick-union	N	N	N
weighted QU	N	$lg\ N$	$lg\ N$

Includes cost of finding roots

Q. Stop here?
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 that root or its grandparent
    - Two-pass implementation: add second loop to root() to set the id[] of each examined node to 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 that root or its grandparent
    - Two-pass implementation: add second loop to root() to set the id[] of each examined node to root
    - Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length)

```
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 completely flat

Weighting & Path Compression

- Weighted quick-union with path compression (WQUPC): amortized analysis
  - Proposition. [Hopcroft Ulman, Tarjan] Starting from an empty data structure, any sequence M union-find operations on N objects makes <= c (N + M *lg** N) array accesses

    Simple algorithm with fascinating mathematics!

  - Linear-time algorithm for M union-find ops on N objets?
    - 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
2^65536	5

Iterate log function

Amazing fact [Fredman-Saks] : No linear-time algorithm exists.

Summary

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

M union-find operations on a set of N objects

Algorithm	Worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M *lg* N
QU + path compression	N + M *lg* N
weighted QU + PC	N + M *lg** N

- Ex. [10^9 unions and finds on 10^9 objects]
  - WQUPC reduces time from 30 years to 6 seconds
  - Supercomputer won't help much; good algorithm enables solution

Outline

- Dynamic Connectivity
- Quick Find
- Quick Union
- Improvements
- Applications

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

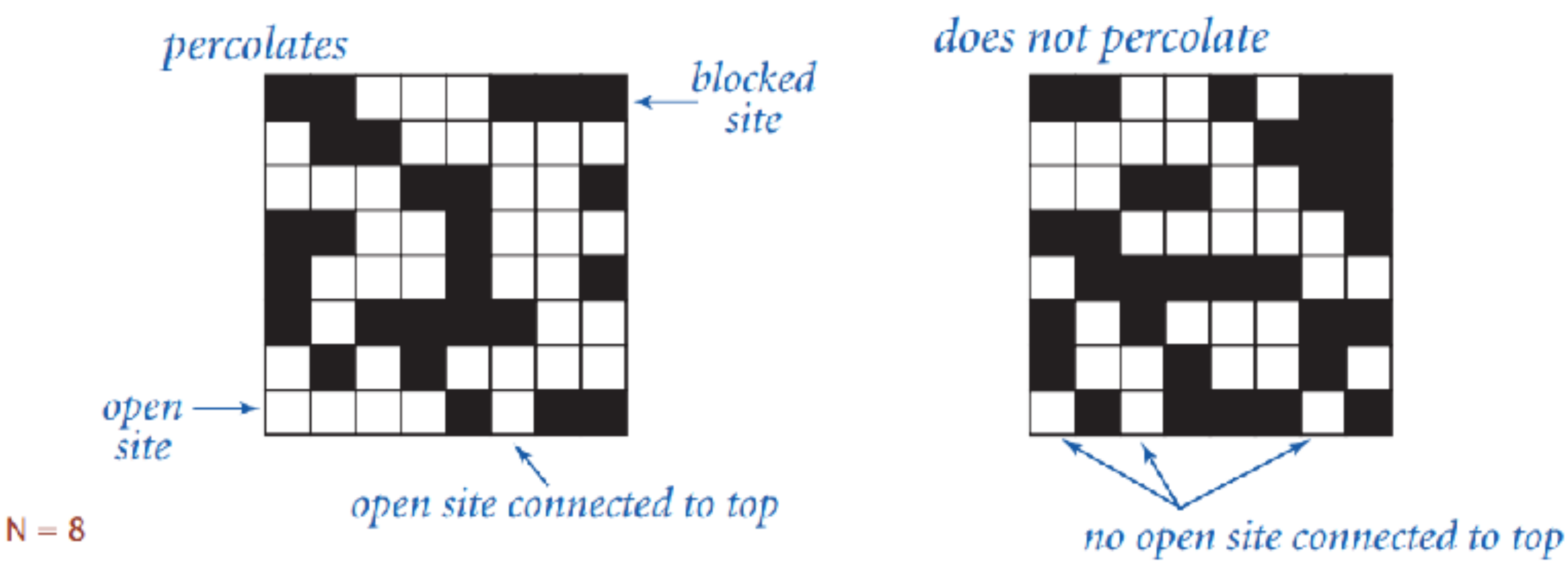
Union-find applications

- Percolation $\boxed{\text{To be introduced}}$
- Games(Go, Hex)
- Dynamic connectivity $\boxed{\text{Done!}}$
- Least common ancestor
- Equivalence of finite state automata
- …

Resources: https://www.coursera.org/learn/algorithms-part1/supplement/bcelg/lecture-slides

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



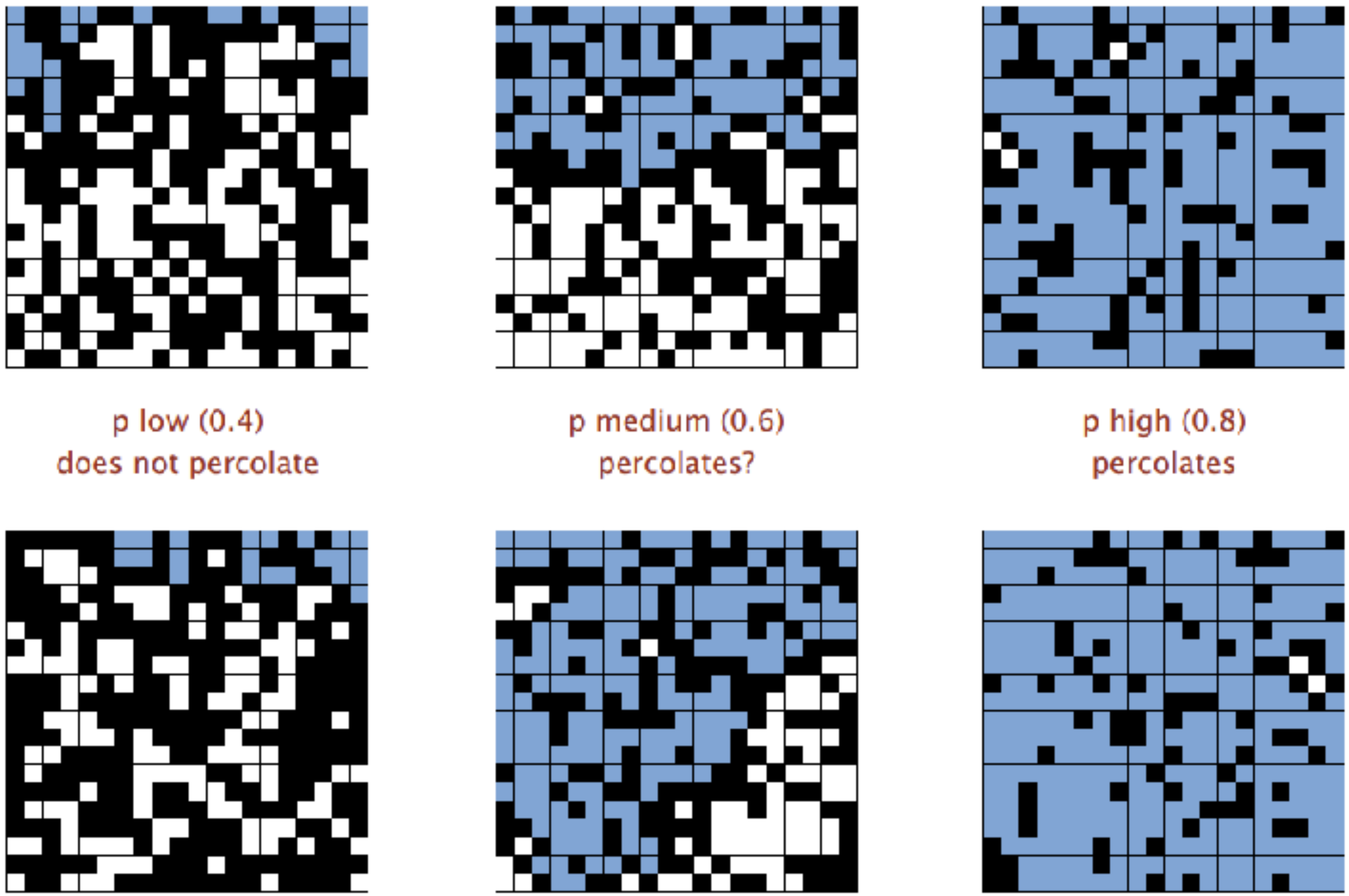
Percolation

- 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

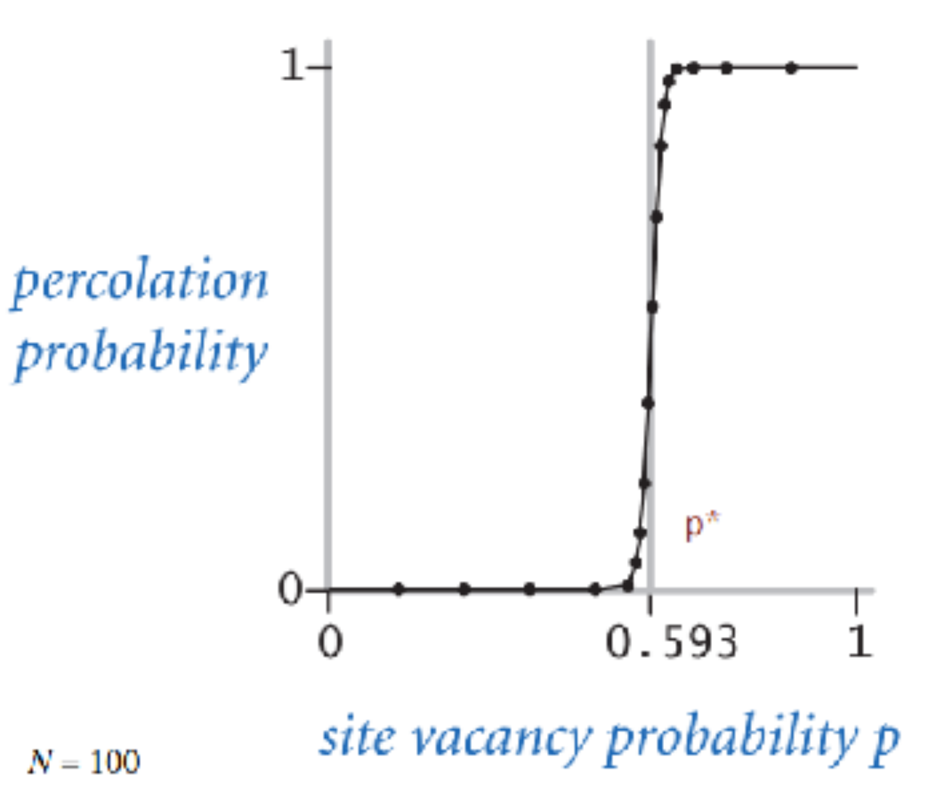
Likelihood of Percolation

- Depends on site vacancy probability p



Percolation Phase Transition

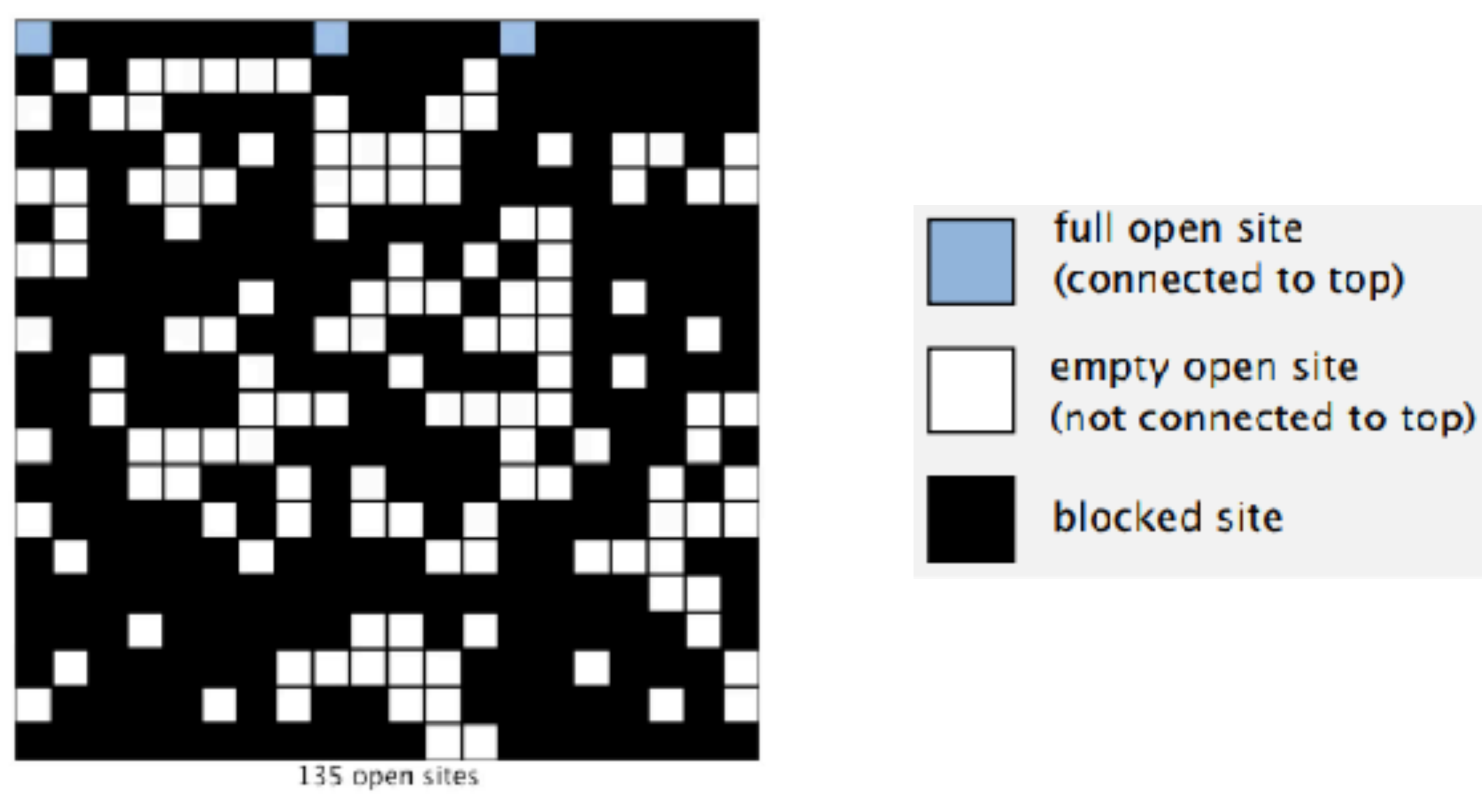
- When N is large, theory guarantees a sharp threshold p^*.
  - $p > p^*$: almost certainly percolates
  - $p < p^*$: almost certainly does not percolates
- Q. What is the value of p^* ?



Solution: Monte Carlo Simulation

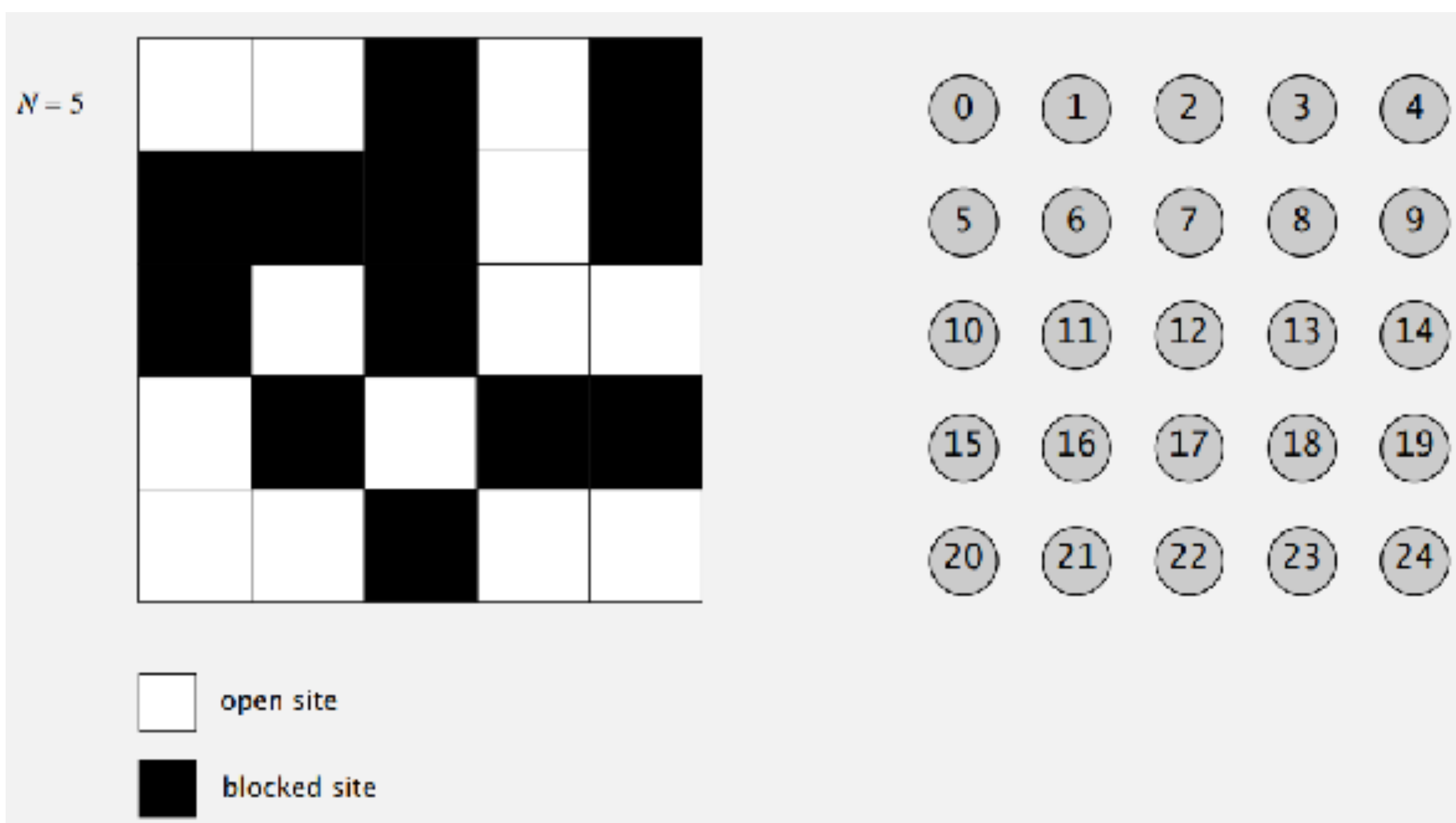
- Initialize N-by-N whole grid to be blocked
- Declare random sites open until top connected to bottom
- Vacancy percentage estimates p^*

N = 20



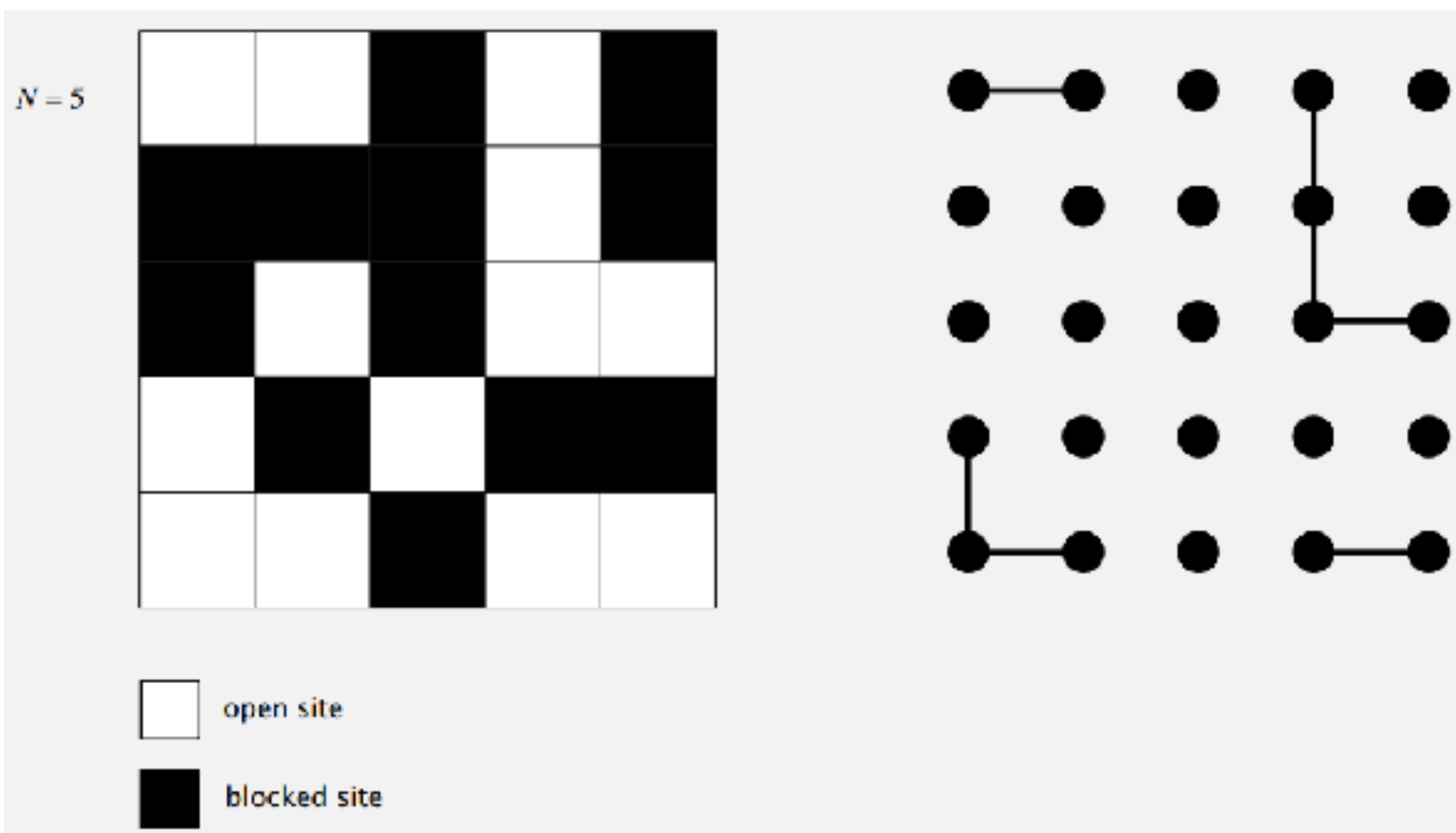
How to Check Percolation?

- Dynamic connectivity solution to estimate percolation threshold
  - Create an object for each site and name them 0 to N^2 -1



How to Check Percolation?

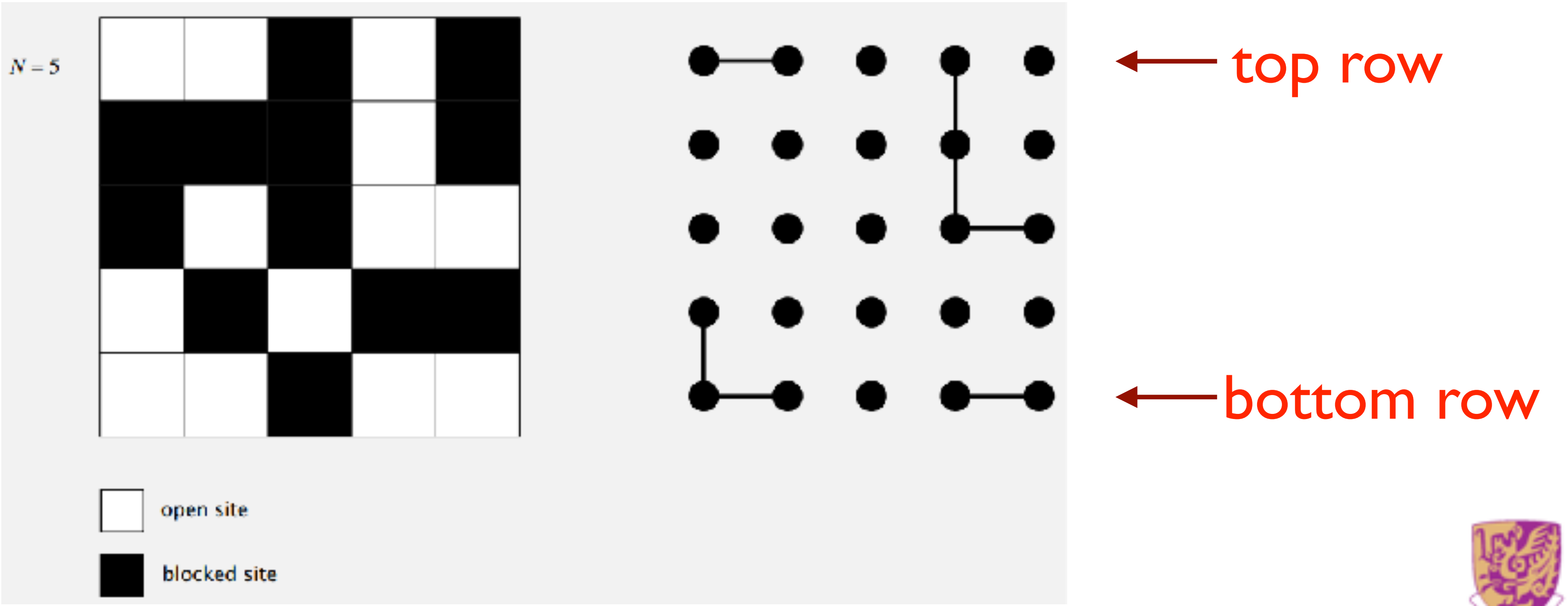
- Dynamic connectivity solution to estimate percolation threshold
  - Create an object for each site and name them 0 to N^2 -1
  - Sites are in same component if connected by open sites.



How to Check Percolation?

- Dynamic connectivity solution to estimate percolation threshold.
  - 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

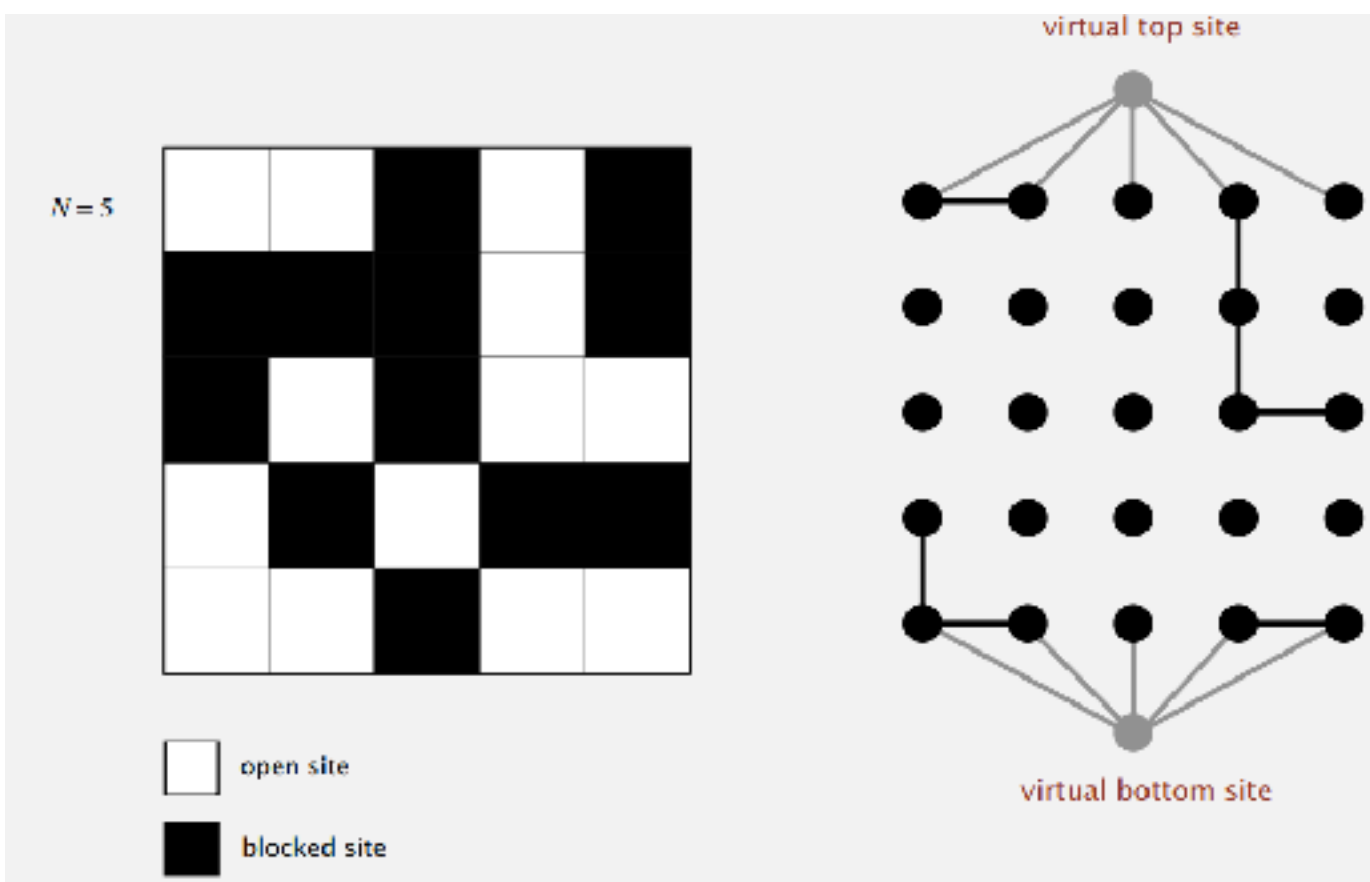
brute-force algorithm: N^2 calls to connected()



How to Check Percolation?

- Dynamic connectivity solution to estimate percolation threshold.
  - Clever trick: Introduce 2 virtual sites (and connections to top and bottom)

    efficient algorithm: only 1 call to connected()
    - Percolates iff virtual top site is connected to virtual bottom site



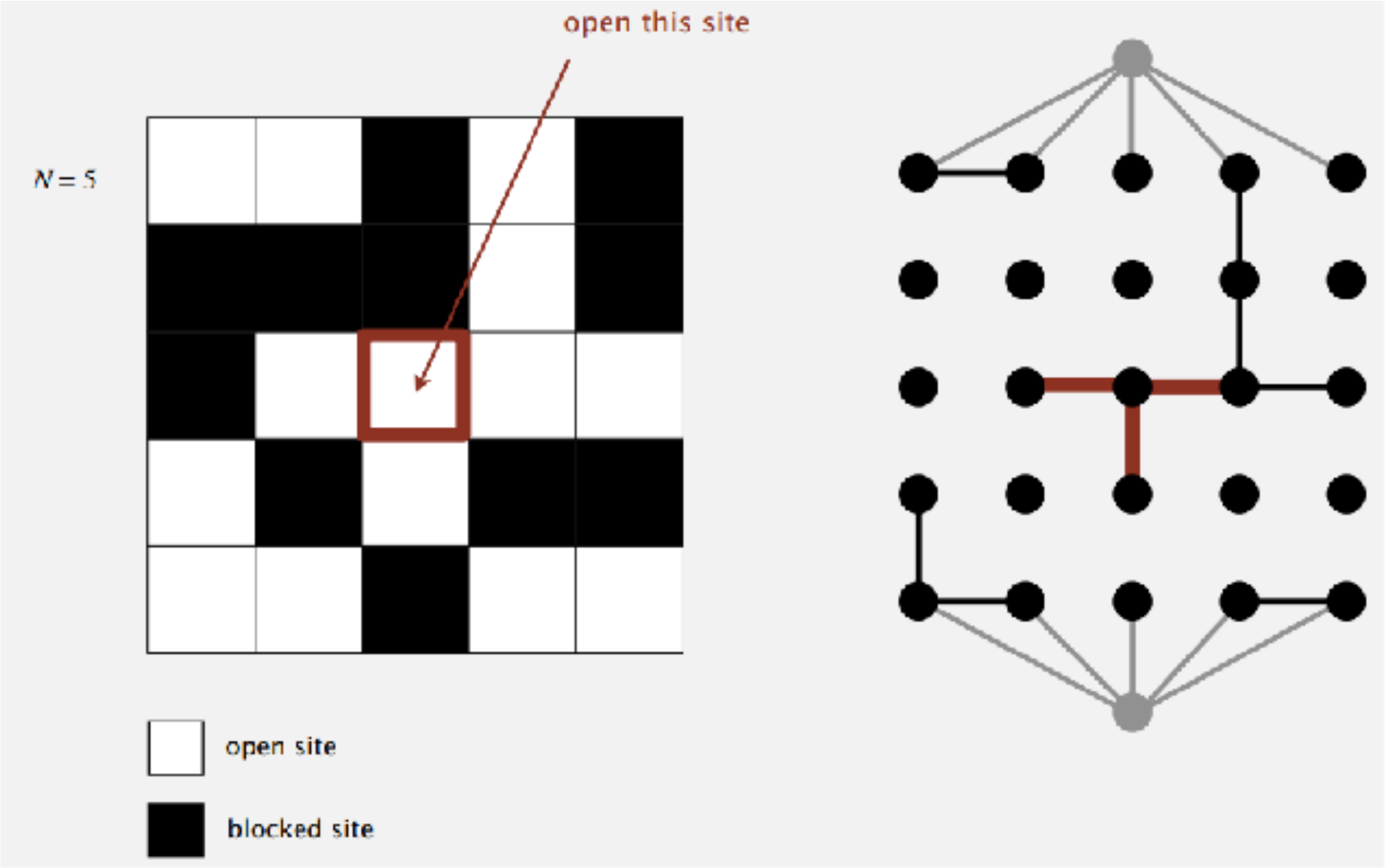
top row

bottom row

How to Model Opening a New Site?

- A. Mark new site as open; connect it to all of its adjacent open sites;

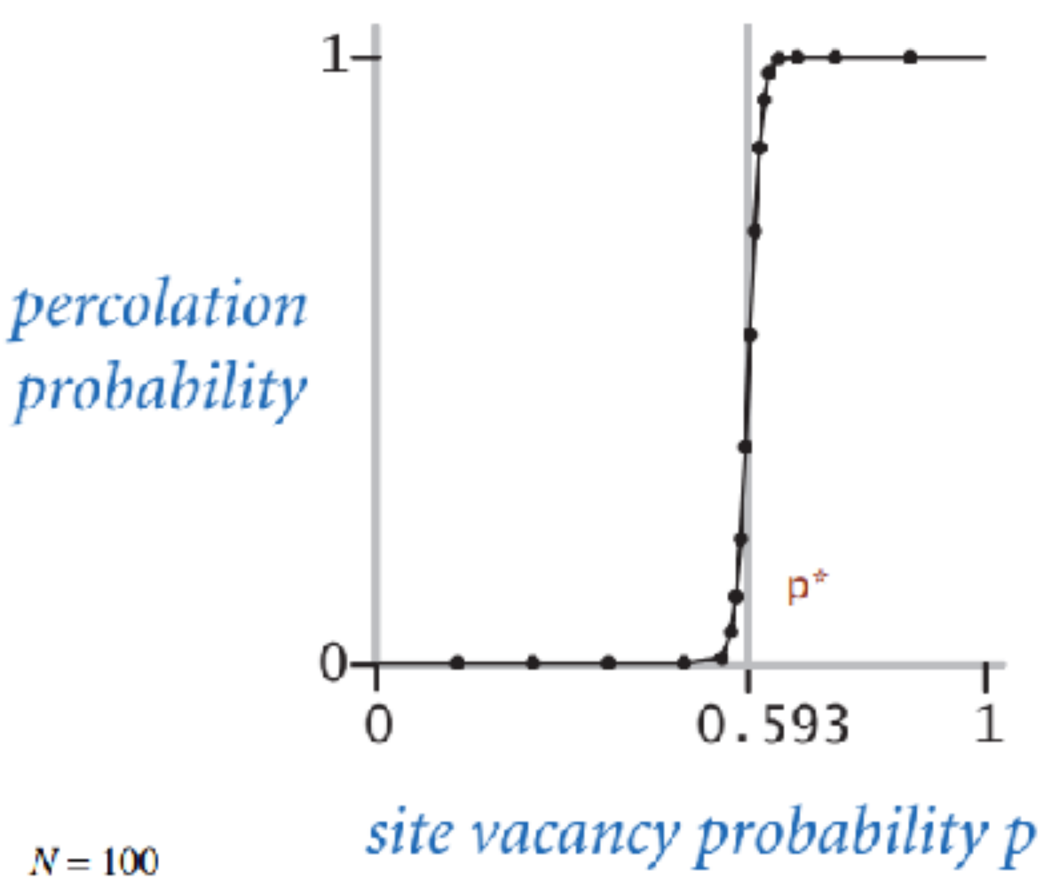
up to 4 calls to union()



Percolation Threshold

- Q. What is percolation threshold p^* ?
- A. About 0.592746 for large square lattice

constant known only via simulation



Fast algorithms enables accurate answer to scientific question.

Purpose

- Learning the 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

Thank You!