ALGORITHMS
AND
DATA STRUCTURES

KEVIN WAYNE

PRINCETON UNIVERSITY

http://www.princeton.edu/~cos226

‣ **outline**
‣ **why study algorithms?**
‣ **usual suspects**
‣ **coursework**
‣ **resources**

COS 226 course overview

What is COS 226?

- Intermediate-level survey course.
- Programming and problem solving, with applications.
- Algorithm: method for solving a problem.
- Data structure: method to store information.

topic	data structures and algorithms
data types	stack, queue, bag, union-find, priority queue
sorting	quicksort, mergesort, heapsort, radix sorts
searching	BST, red-black BST, hash table
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	KMP, regular expressions, TST, Huffman, LZW
advanced	B-tree, suffix array, maxflow, simplex

Why study algorithms?

Their impact is broad and far-reaching.

Internet. Web search, packet routing, distributed file sharing, ...

Biology. Human genome project, protein folding, ...

Computers. Circuit layout, file system, compilers, ...

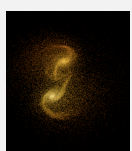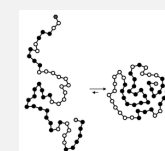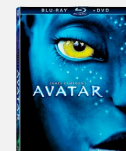Computer graphics. Movies, video games, virtual reality, ...

Security. Cell phones, e-commerce, voting machines, ...

Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

Social networks. Recommendations, news feeds, advertisements, ...

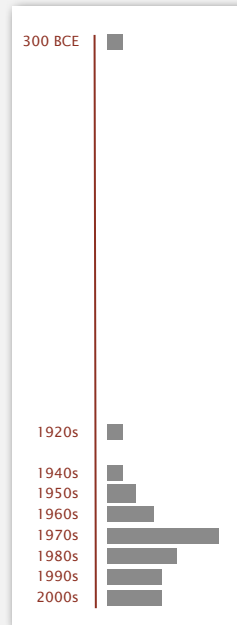Physics. N-body simulation, particle collision simulation, ...

...

Why study algorithms?

Old roots, new opportunities.

- Study of algorithms dates at least to Euclid.
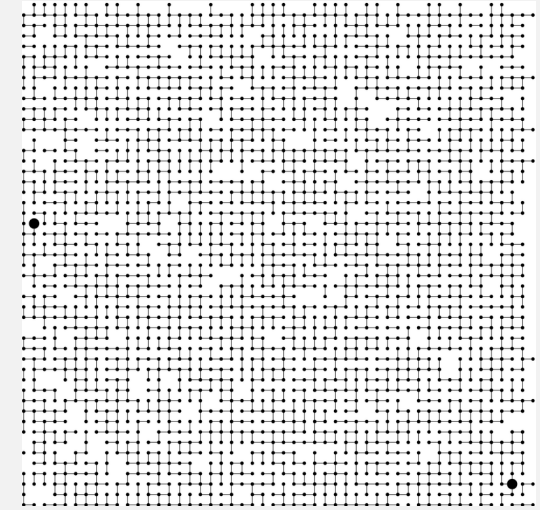- Some important algorithms were discovered by undergraduates in a course like this!

300 BCE

1920s

1940s
1950s
1960s
1970s
1980s
1990s
2000s

Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity. [stay tuned]

Why study algorithms?

For intellectual stimulation.

" For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. "   — F. Sullivan

" An algorithm must be seen to be believed. "   — D. E. Knuth

---

Why study algorithms?

To become a proficient programmer.

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "
    — Linus Torvalds (creator of Linux)

" Algorithms + Data Structures = Programs. "   — Niklaus Wirth

Algorithms +
Data
Structures =
Programs

Why study algorithms?

They may unlock the secrets of life and of the universe.

Computational models are replacing mathematical models in scientific inquiry.

$$E = mc^2$$
$$F = ma$$
$$F = \frac{Gm_1 m_2}{r^2}$$
$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(r) \right] \Psi(r) = E\,\Psi(r)$$

```
for (double t = 0.0; true; t = t + dt)
    for (int i = 0; i < N; i++)
    {
        bodies[i].resetForce();
        for (int j = 0; j < N; j++)
            if (i != j)
                bodies[i].addForce(bodies[j]);
    }
```

20th century science
(formula based)

21st century science
(algorithm based)

" *Algorithms: a common language for nature, human, and computer.* " — *A. Wigderson*

Why study algorithms?

For fun and profit.

Why study algorithms?

- Their impact is broad and far-reaching.
- Old roots, new opportunities.
- To solve problems that could not otherwise be addressed.
- For intellectual stimulation.
- To become a proficient programmer.
- They may unlock the secrets of life and of the universe.
- For fun and profit.

Why study anything else?

The usual suspects

Lectures. Introduce new material.

Precepts. Discussion, problem-solving, background for programming assignment.

What	When	Where	Who	Office Hours
L01	TTh 11–12:20	CS 104	Kevin Wayne	see web
P01	F 11–11:50	Friend 112	Maia Ginsburg †	see web
P01A	F 11–11:50	Friend 108	Aman Dhesi	see web
P02	F 12:30–1:20	Friend 112	Joey Dodds	see web
P02A	F 12:30–1:20	Friend 108	Steven Liu	see web
P03	F 1:30–2:20	Friend 112	Maia Ginsburg †	see web
P03A	F 1:30–2:20	Friend 108	Sasha Koruga	see web

† lead preceptor

Where to get help?

Piazza. Online discussion forum.
- Short questions.
- Clarifications on lectures, readings, and assignments.

piazza

http://www.piazza.com/class#cos226fall2011

Email. For personal (or solution-revealing) questions.

Gmail

YAHOO! Mail

Office hours. For longer questions.

open

Computing laboratory. Undergrad TAs in Friend 016. See web for schedule.

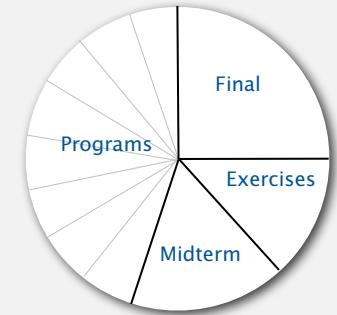## Coursework and grading

Programming assignments. 45%
Due at 11pm via electronic submission.

Written exercises. 15%
Due at 11am in lecture.

Exams. 15% + 25%
- Closed-book with cheatsheet.
- Midterm (in class on Tuesday, October 25).
- Final (scheduled by Registrar).
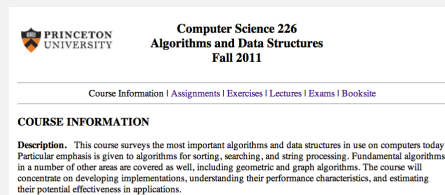
Staff discretion. To adjust borderline cases.
- Report errata.
- Contribute to Piazza discussions.
- Attend and participate in precept/lecture.

Final
Programs
Exercises
Midterm

Resources (web)

Course content.
- Course info.
- Programming assignments.
- Exercises.
- Lecture slides.
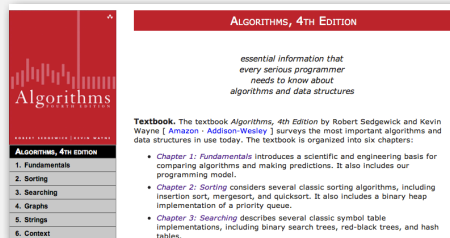- Exam archive.
- Submit assignments.

PRINCETON UNIVERSITY
Computer Science 226
Algorithms and Data Structures
Fall 2011

Course Information | Assignments | Exercises | Lectures | Exams | Booksite

COURSE INFORMATION

Description. This course surveys the most important algorithms and data structures in use on computers today. Particular emphasis is given to algorithms for sorting, searching, and string processing. Fundamental algorithms in a number of other areas are covered as well, including geometric and graph algorithms. The course will concentrate on developing implementations, understanding their performance characteristics, and estimating their potential effectiveness in applications.

http://www.princeton.edu/~cos226

Booksites.
- Brief summary of content.
- Download code from lecture.

ALGORITHMS, 4TH EDITION

essential information that every serious programmer needs to know about algorithms and data structures

Textbook. The textbook *Algorithms, 4th Edition* by Robert Sedgewick and Kevin Wayne [Amazon · Addison-Wesley] surveys the most important algorithms and data structures in use today. The textbook is organized into six chapters:

ALGORITHMS, 4TH EDITION
1. Fundamentals
2. Sorting
3. Searching
4. Graphs
5. Strings
6. Context

- *Chapter 1: Fundamentals* introduces a scientific and engineering basis for comparing algorithms and making predictions. It also includes our programming model.
- *Chapter 2: Sorting* considers several classic sorting algorithms, including insertion sort, mergesort, and quicksort. It also includes a binary heap implementation of a priority queue.
- *Chapter 3: Searching* describes several classic symbol table implementations, including binary search trees, red-black trees, and hash tables.

http://www.algs4.princeton.edu

Resources (textbook)

Required readings: Algorithms 4th edition by R. Sedgewick and K. Wayne, Addison-Wesley Professional, 2011, ISBN 0-321-57351-X.

Algorithms
FOURTH EDITION
ROBERT SEDGEWICK | KEVIN WAYNE

Available in hardcover and Kindle.
- Online: Amazon ($60 to buy), Chegg ($40 to rent), ...
- Brick-and-mortar: Labyrinth Books (122 Nassau St).
- On reserve: Engineering library.

What's ahead?

Lecture 1. Union find. ← today
Precept 1. Meets tomorrow.
Lecture 2. Analysis of algorithms.

Exercise 1. Due via hardcopy in lecture at 11am on Tuesday.
Assignment 1. Due via electronic submission at 11pm on Wednesday.

Right course? See me.
Placed out of COS 126? Review Sections 1.1–1.2 of Algorithms, 4th edition
(includes command-line interface and our I/O libraries).

Not registered? Go to any precept tomorrow.
Change precept? Use SCORE. ← see Colleen Kenny-McGinley in CS 210
if the only precept you can attend is closed

1.5 UNION FIND

Algorithms
FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

‣ 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.

‣ dynamic connectivity
‣ quick find
‣ quick union
‣ improvements
‣ applications
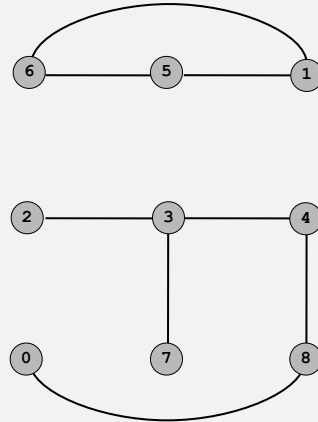
Given a set of objects

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

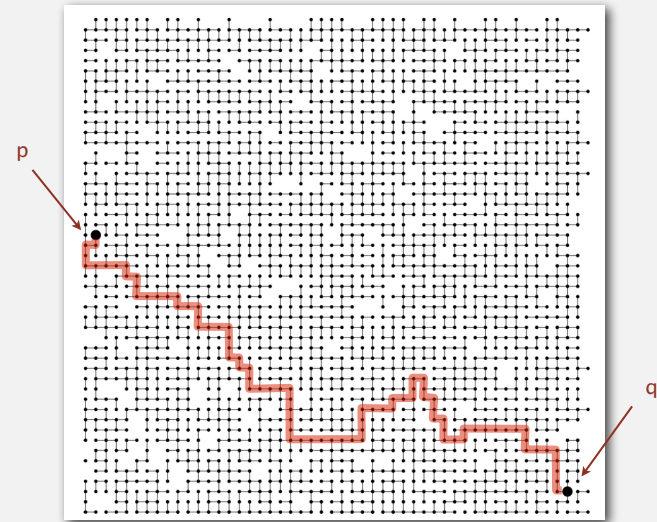more difficult problem: find the path

```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
connected(0, 2)   no
connected(2, 4)   yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
connected(0, 2)   yes
connected(2, 4)   yes
```


4

Q. Is there a path from p to q?

A. Yes.

5

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)
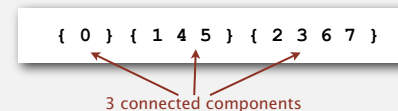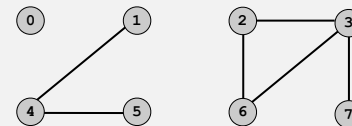
6

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.

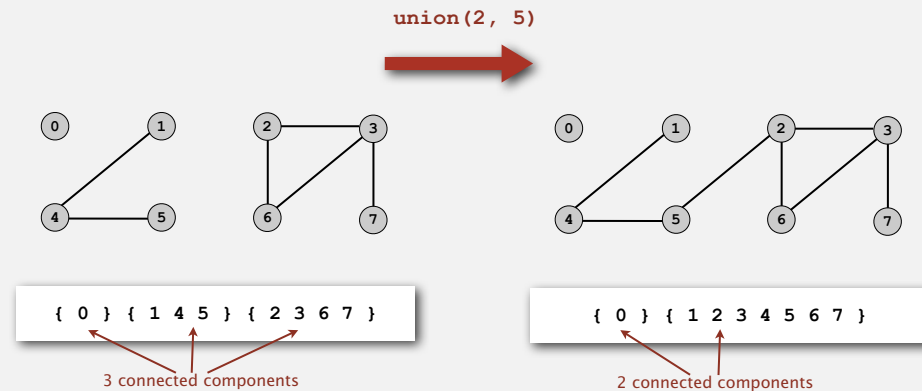{ 0 } { 1 4 5 } { 2 3 6 7 }

3 connected components

7

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(2, 5)

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

3 connected components

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

2 connected components

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
  - write out pair if they are not already connected

```
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
```
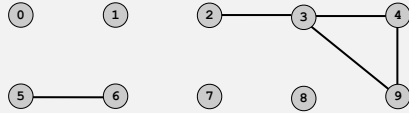
Quick-find [eager approach]

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

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

5 and 6 are connected
2, 3, 4, and 9 are connected

(graph: nodes 0, 1; nodes 2—3, 3—4, 3—9, 4—9 connected; nodes 5—6 connected; nodes 7, 8)

Quick-find [eager approach]

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

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

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

`id[3] = 9; id[6] = 6`
3 and 6 are not connected

Quick-find [eager approach]

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

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

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

`id[3] = 9; id[6] = 6`
3 and 6 are not connected

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

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

after union of 3 and 6

problem: many values can change

Quick-find example

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

id[]

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*

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 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;
    }
}
```

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

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

change all entries with `id[p]` to `id[q]`
(linear number of array accesses)

Quick-find is too slow

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

algorithm	init	union	connected
quick-find	N	N	1

order of growth of number of array accesses

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.

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!
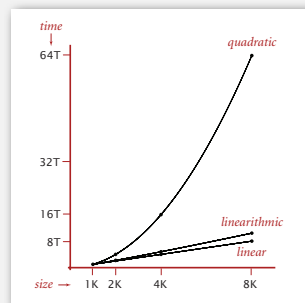
Ex. Huge problem for quick-find.

- 10^9 union commands on 10^9 objects.
- Quick-find takes more than 10^{18} operations.
- 30+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.

- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
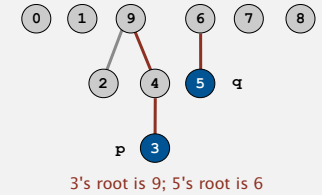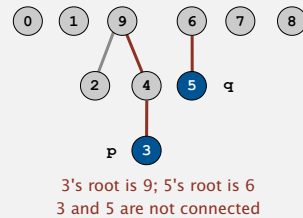- With quadratic algorithm, takes 10x as long!

Quick-union [lazy approach]

Data structure.

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

keep going until it doesn't change

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

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

Quick-union [lazy approach]

Data structure.

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

keep going until it doesn't change

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

Find. Check if `p` and `q` have the same root.

3's root is 9; 5's root is 6
3 and 5 are not connected

Quick-union [lazy approach]

Data structure.

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

keep going until it doesn't change

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

Find. Check if `p` and `q` have the same root.

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

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

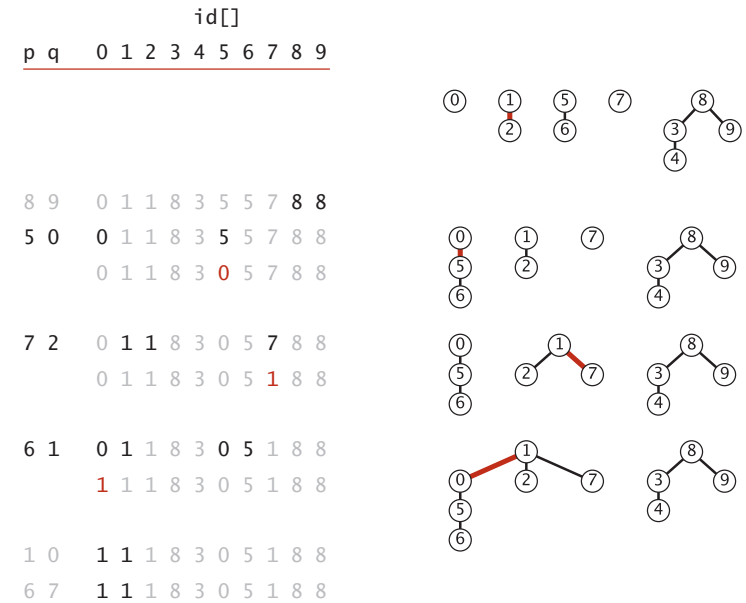only one value changes
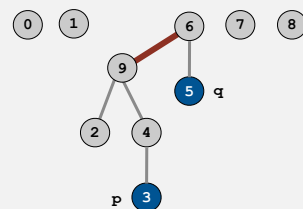
---

Quick-union demo
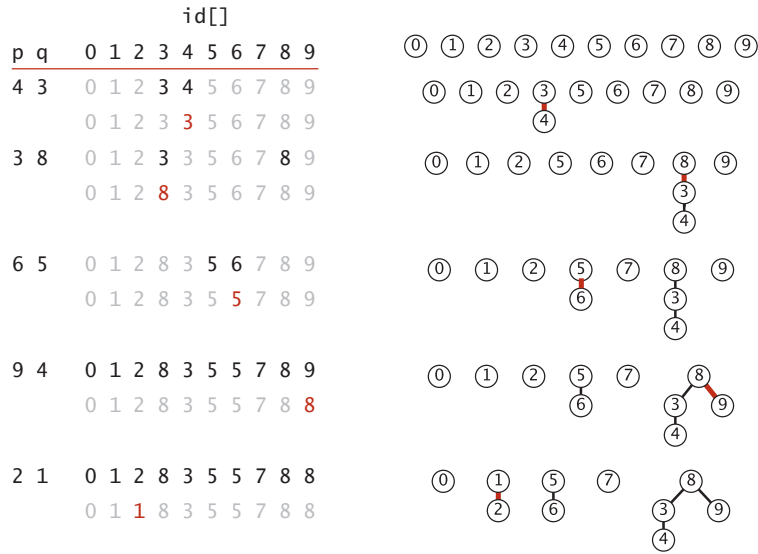
Quick-union example

Quick-union example

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;
    }

    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);
        int 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).

algorithm	init	union	connected
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 them flat.

Quick-union defect.
- Trees can get tall.
- Find too expensive (could be N array accesses).

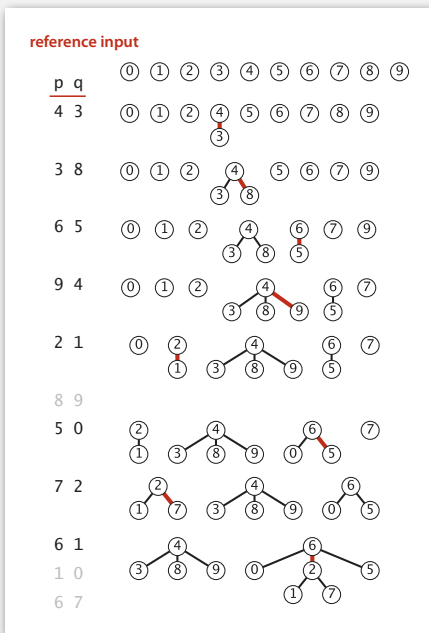## Slide 28

28

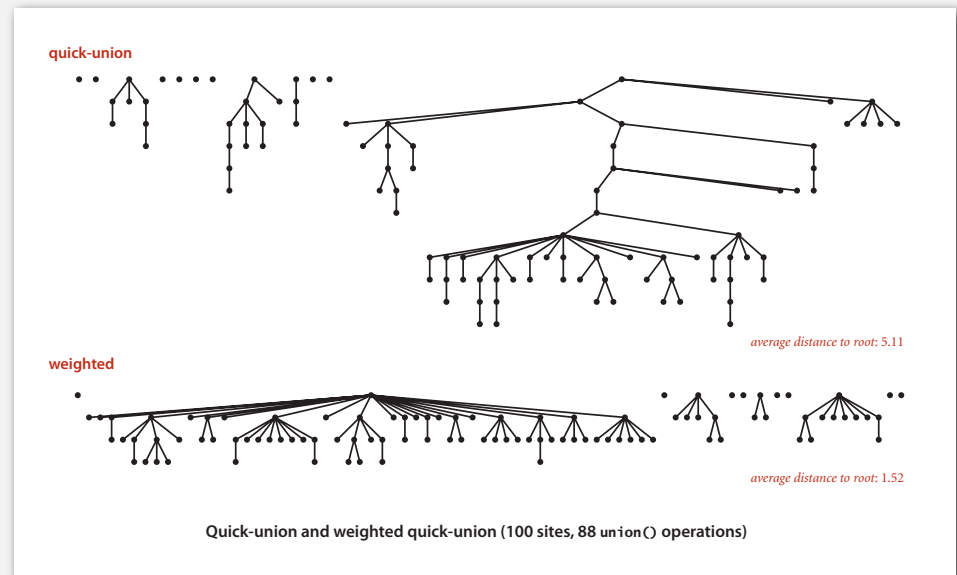## Slide 29

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.
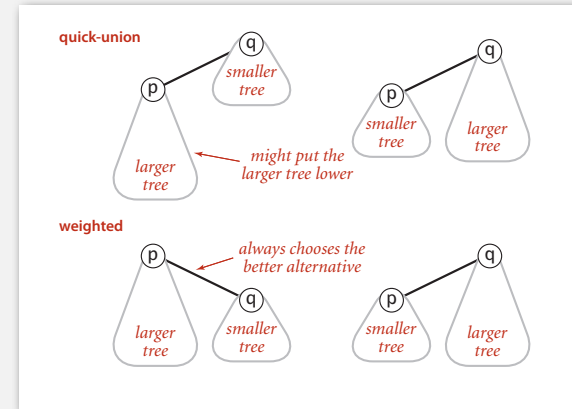
29

Slide 30

Weighted quick-union examples

30

Slide 31

Quick-union and weighted quick-union example

average distance to root: 5.11

average distance to root: 1.52

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

31
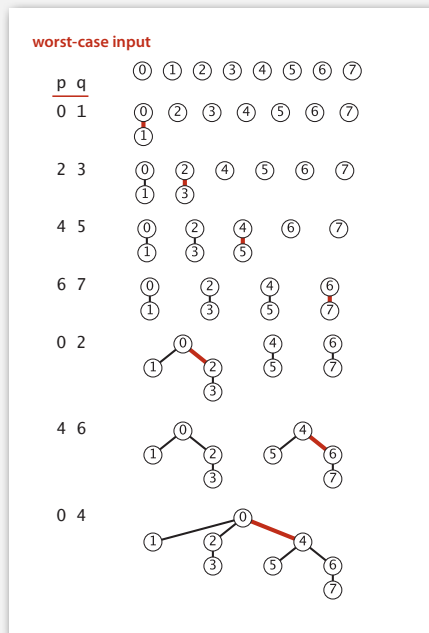
Weighted quick-union: Java implementation

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:
- Merge smaller tree into larger tree.
- Update the `sz[]` array.

```
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]; }
```

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 $\lg N$.

N = 10
depth(x) = 3 ≤ lg N
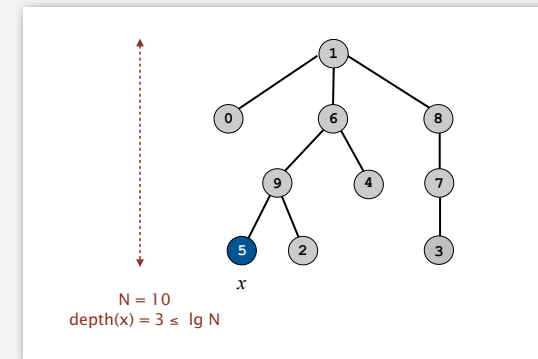
Weighted quick-union analysis

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?
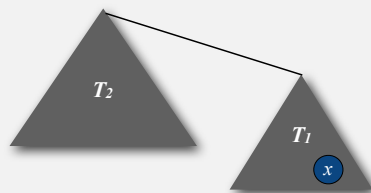
Weighted quick-union analysis

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 $\lg N$.

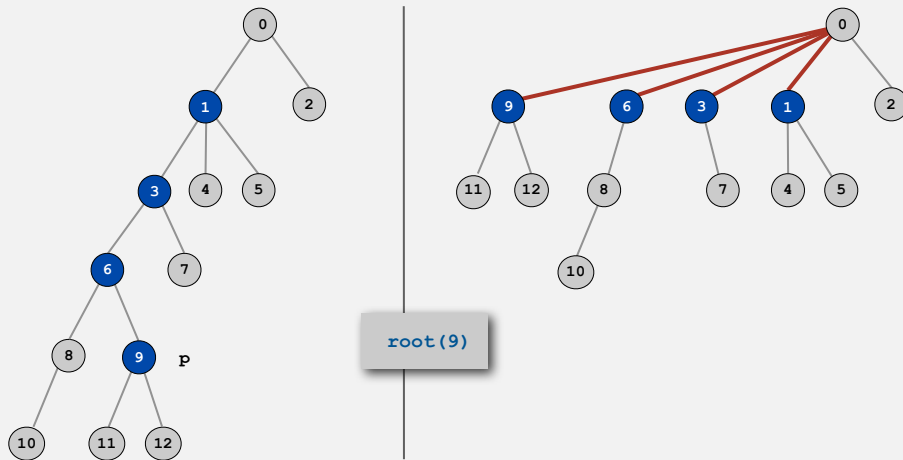| algorithm	init	union	connected
quick–find	N	N	1
quick–union	N	N †	N
weighted QU	N	lg N †	lg N

† 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.

root(9)

Path compression: Java implementation

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

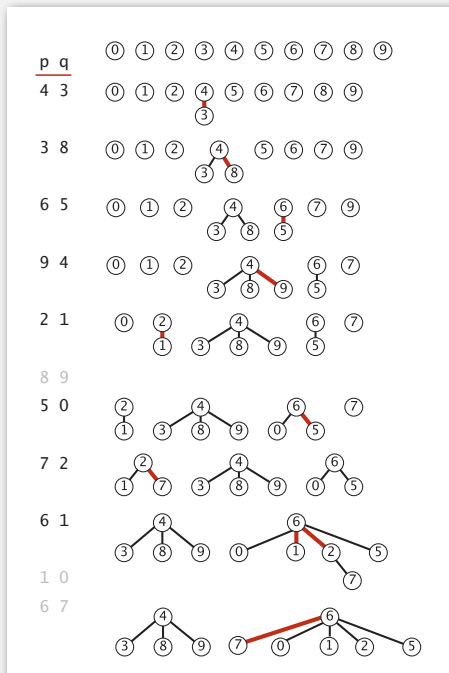Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length).

```
private int root(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]];        ← only one extra line of code !
        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

Weighted quick-union with path compression: amortized analysis

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.
- But the algorithm is still simple!   see COS 423
- Analysis can be improved to $N + M \alpha(M, N)$.

**Bob Tarjan
(Turing Award '86)**

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.

because $\lg^* N$ is a constant in this universe

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

lg* function

Amazing fact. No linear-time algorithm exists.

in "cell-probe" model of computation

Summary

Bottom line. WQUPC 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.

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.
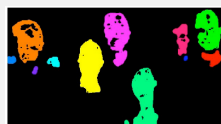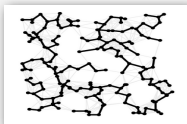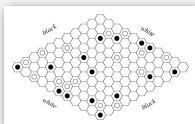
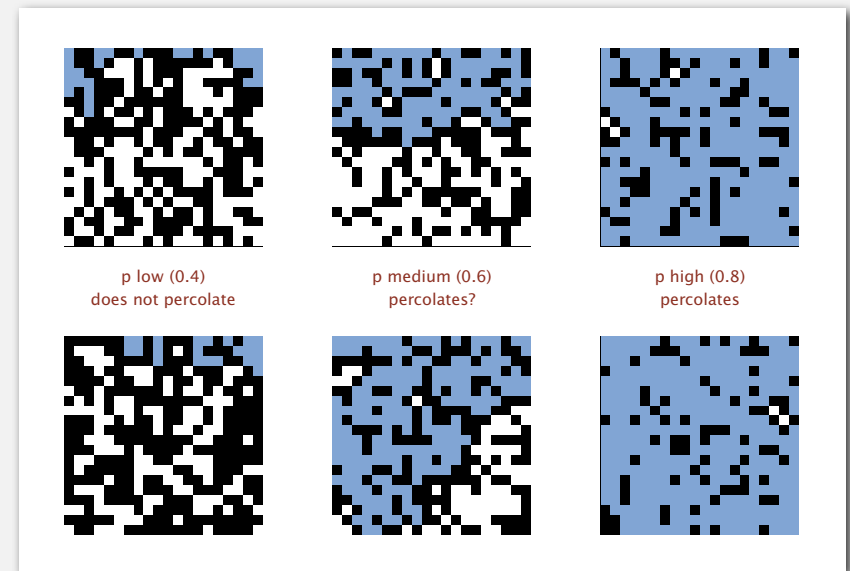## Percolation

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.

percolates   *does not percolate*

blocked site

full open site

empty open site

open site connected to top

no open site connected to top

$N = 8$
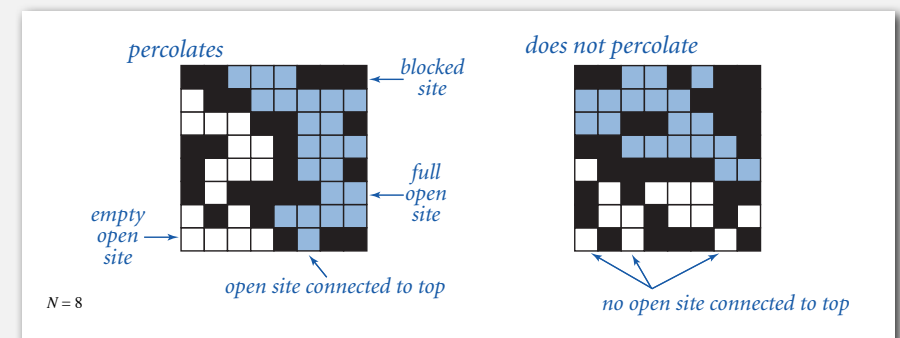
Percolation

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^* ?

percolation probability

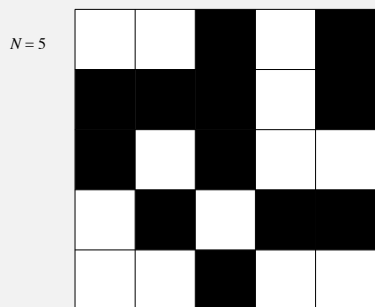*site vacancy probability p*

$N = 100$
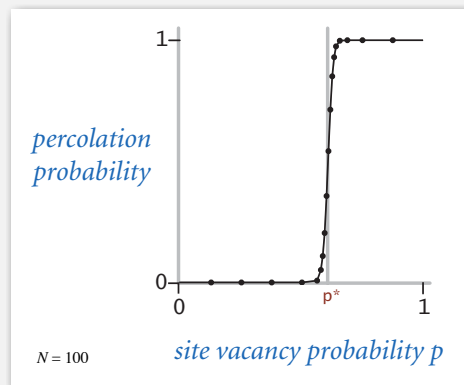
Monte Carlo simulation

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

full open site
(connected to top)

empty open site
(not connected to top)

blocked site

$N = 20$

135 open sites

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

$N = 5$

open site

blocked site

48

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

• Create an object for each site and name them 0 to $N^2 - 1$.

$N = 5$

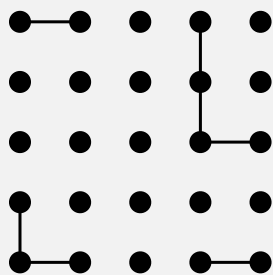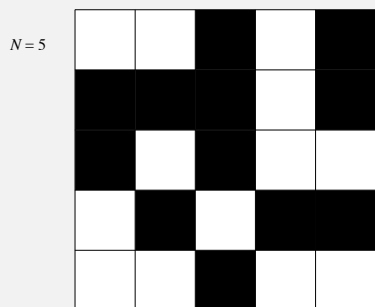open site

blocked site

49

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.

$N = 5$

open site

blocked site

50

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()

$N = 5$
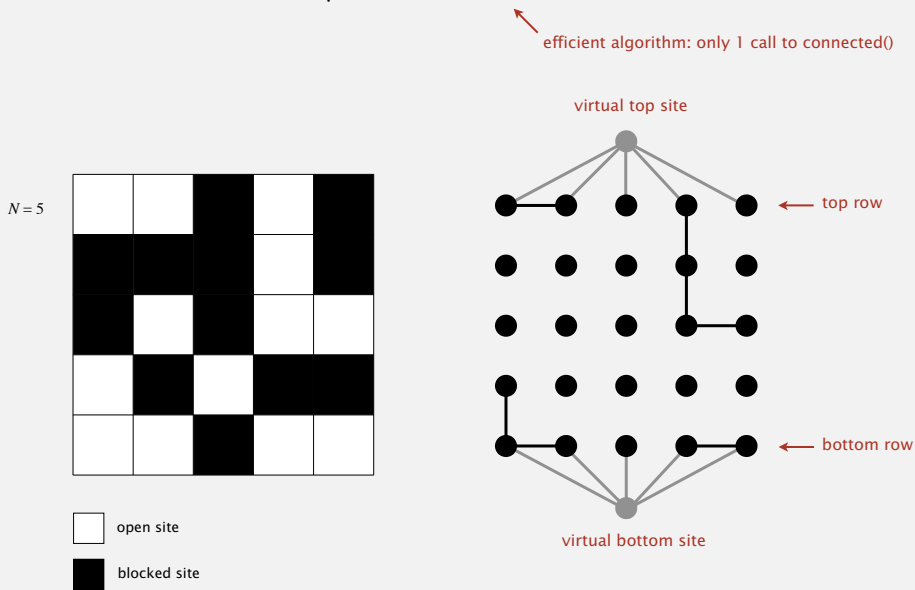

← top row

← bottom row

open site

blocked site
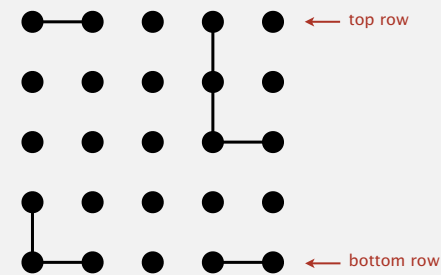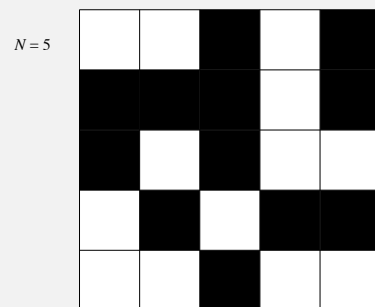
51

Dynamic connectivity solution to estimate percolation threshold

Clever trick. Introduce two 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()

virtual top site

← top row

← bottom row

virtual bottom site

□ open site

■ blocked site

52

Dynamic connectivity solution to estimate percolation threshold

Q. How to model as dynamic connectivity problem when opening a new site?
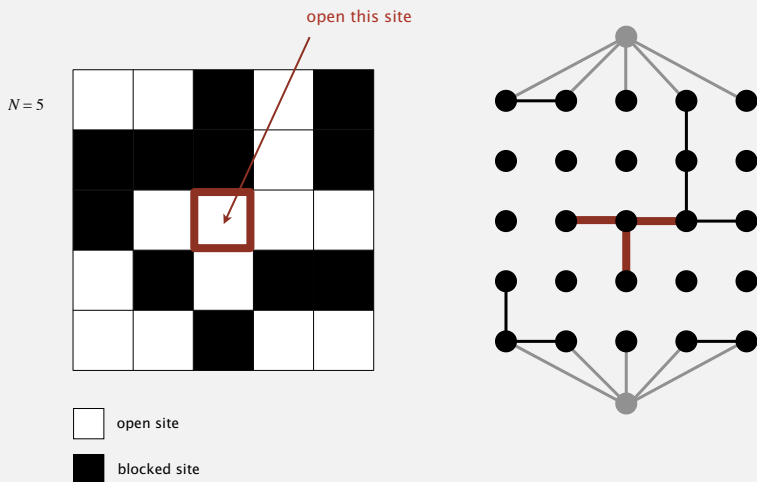
open this site

□ open site

■ blocked site
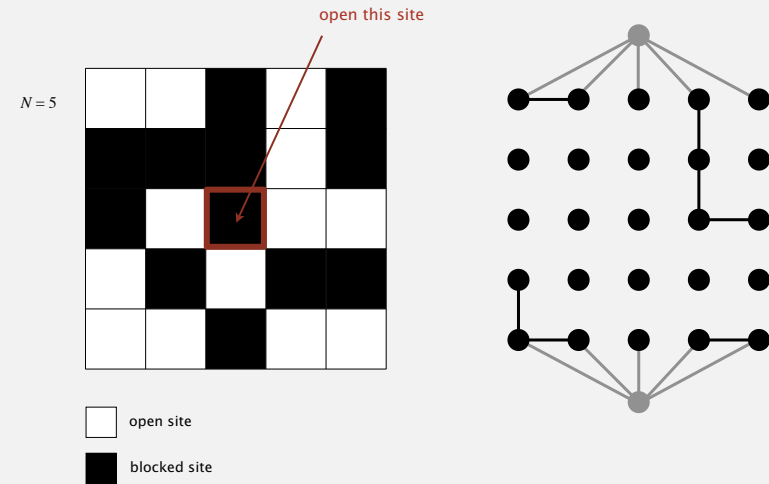
53

Dynamic connectivity solution to estimate percolation threshold

Q. How to model as dynamic connectivity problem when opening a new site?

A. Connect newly opened site to all of its adjacent open sites.
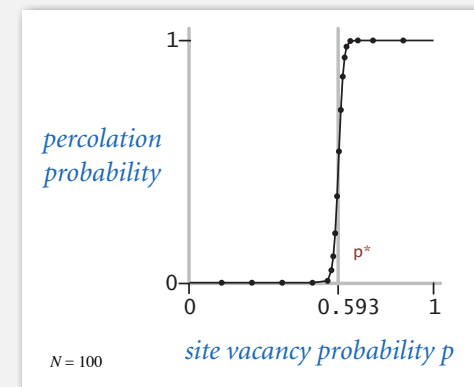
up to 4 calls to union()

open this site

□ open site

■ blocked site

54

Percolation threshold

Q. What is percolation threshold $p*$?

A. About 0.592746 for large square lattices.

constant know only via simulation

percolation probability

$p*$

0.593

site vacancy probability p

$N = 100$

Fast algorithm enables accurate answer to scientific question.

55

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.