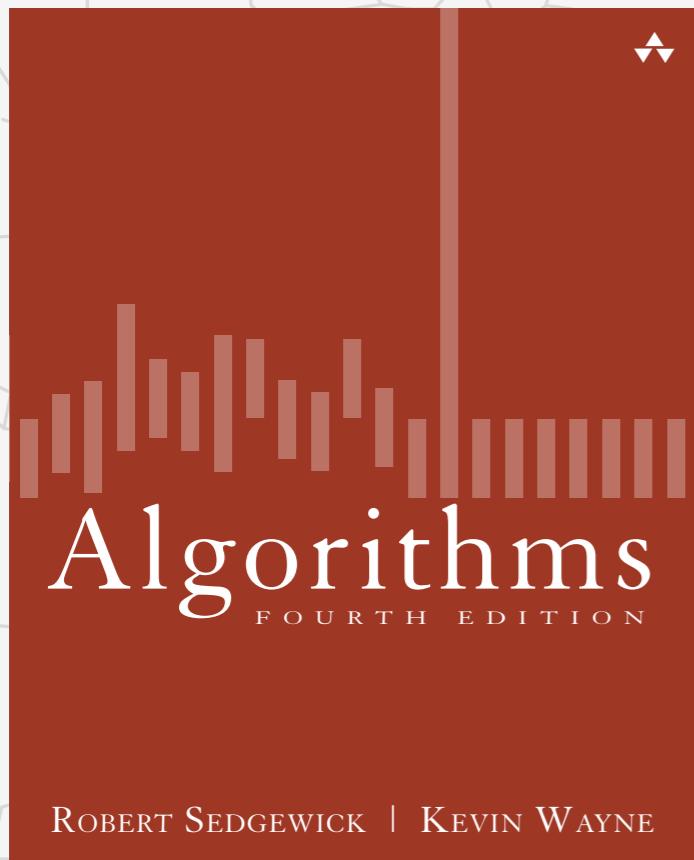


Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ALGORITHMS, PARTS I AND II

- ▶ **overview**
- ▶ ***why study algorithms?***
- ▶ **resources**

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

Course overview

What is this course?

- 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	part 1
searching	BST, red-black BST, hash table	
graphs	BFS, DFS, Prim, Kruskal, Dijkstra	
strings	radix sorts, tries, KMP, regexps, data compression	part 2
advanced	B-tree, suffix array, maxflow	

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, ...

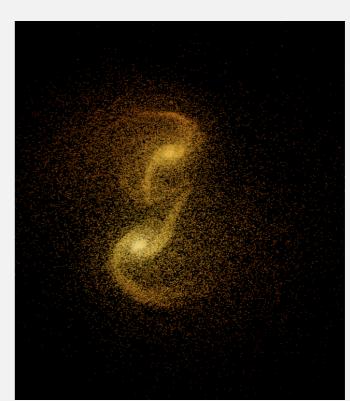
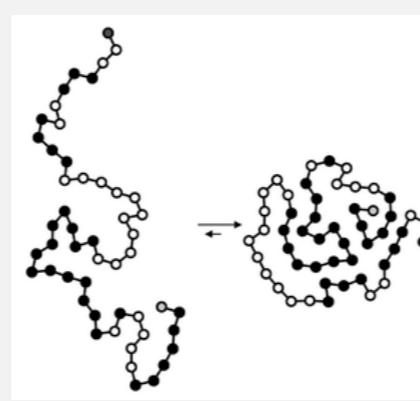
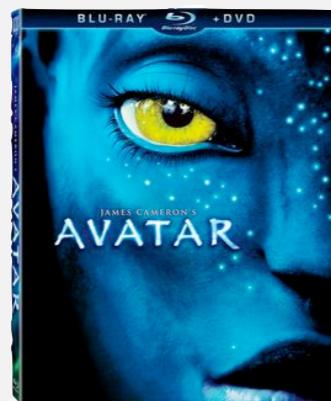
Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

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

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

:

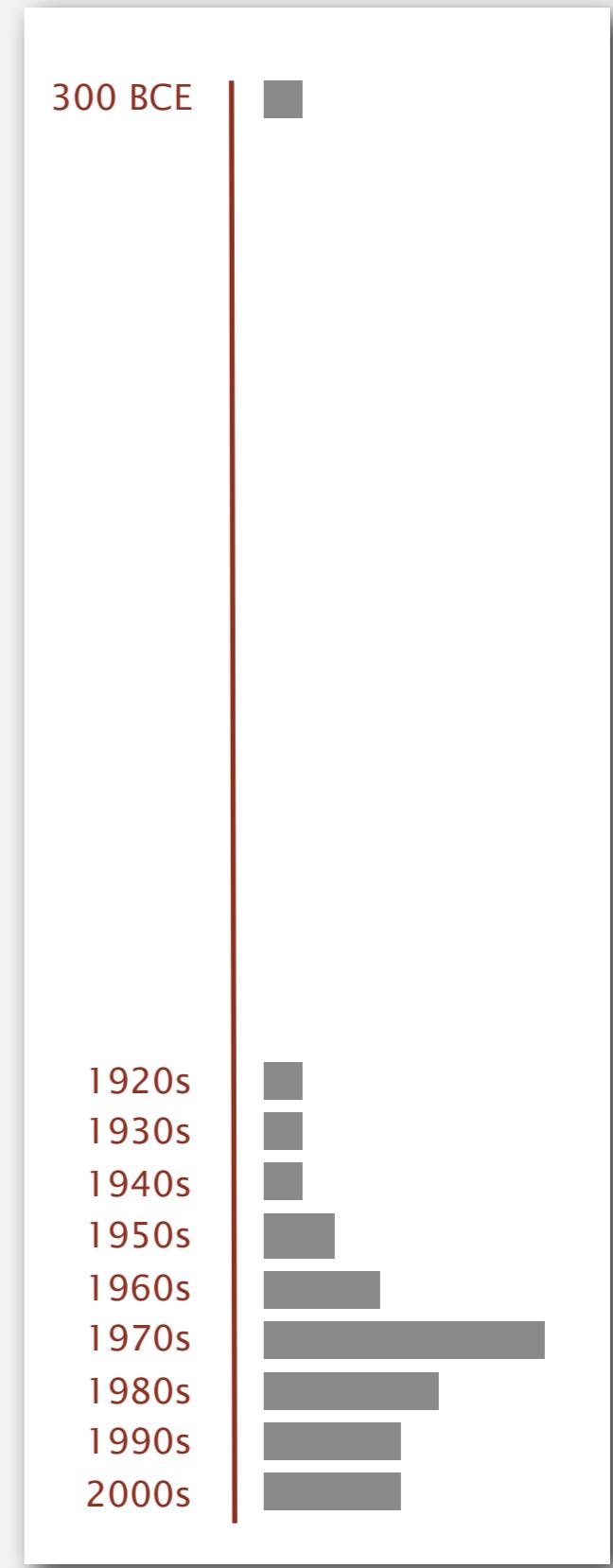
Google
YAHOO!
bing™



Why study algorithms?

Old roots, new opportunities.

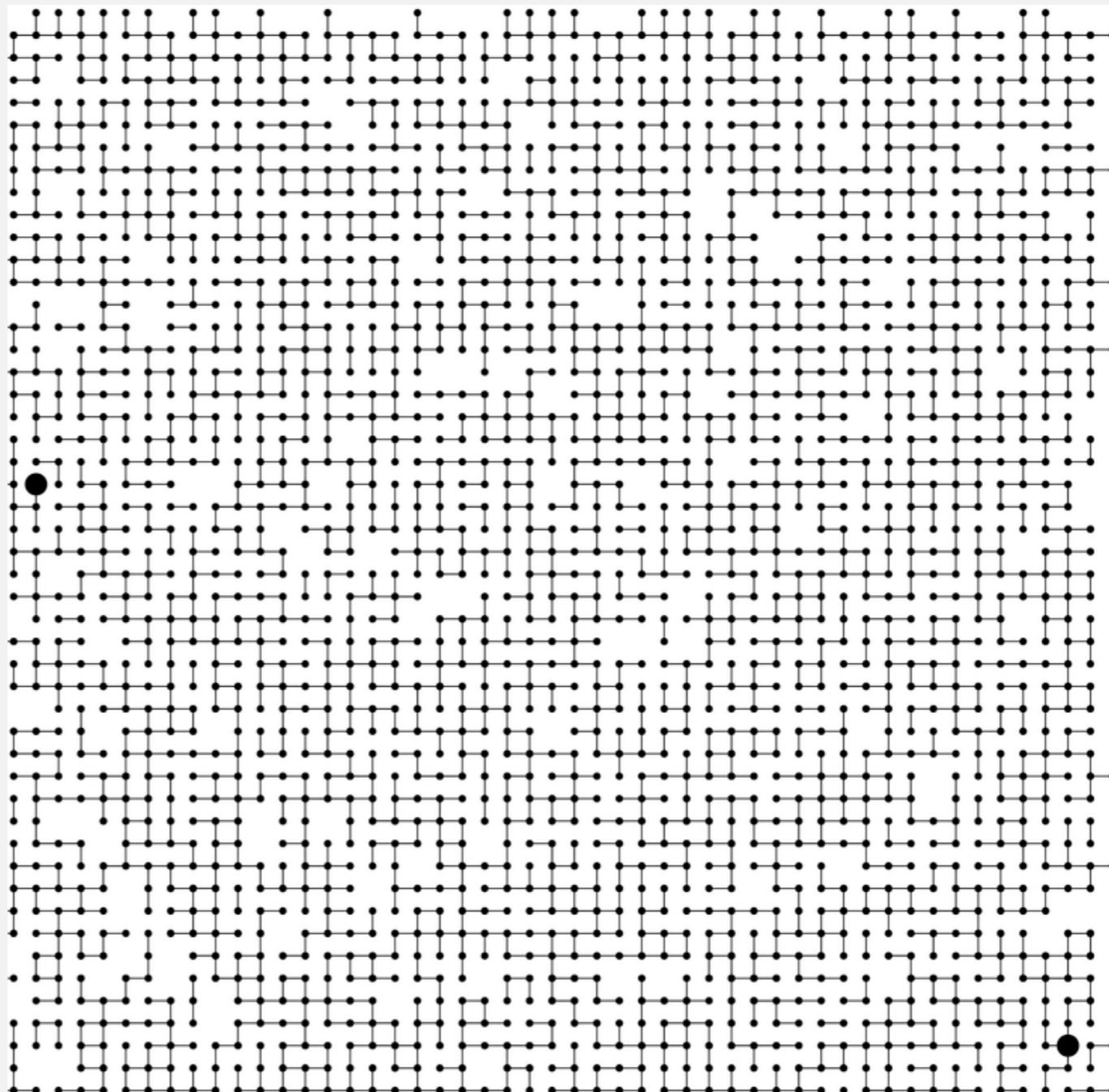
- Study of algorithms dates at least to Euclid.
- Formalized by Church and Turing in 1930s.
- Some important algorithms were discovered by undergraduates in a course like this!



Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity. [stay tuned]

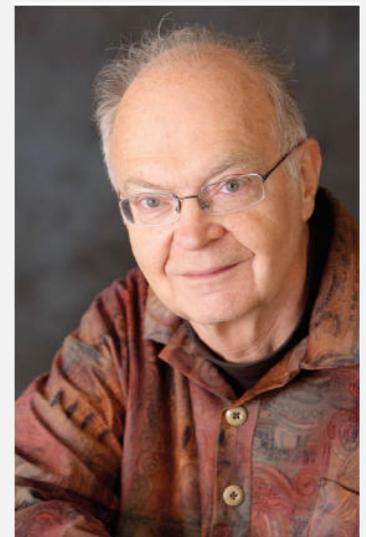


Why study algorithms?

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.” — Francis Sullivan

“ An algorithm must be seen to be believed. ” — Donald Knuth



FROM THE
EDITORS

THE JOY OF ALGORITHMS

Frances Sullivan, Associate Editor-in-Chief

THE THEME OF THIS FIRST-OF-THE-CENTURY ISSUE OF COMPUTING IN SCIENCE & ENGINEERING IS ALGORITHMS. IN FACT, WE WERE BOLD ENOUGH—AND PERHAPS FOOLISH ENOUGH—TO CALL THE 10 EXAMPLES WE’VE SELECTED “THE TOP 10 ALGORITHMS OF THE CENTURY.”

Computational algorithms are probably as old as civilization. Sumerian cuneiform, one of the most ancient written records, contains partly of algebraic descriptions for computing in base 60. And the Chinese abacus, invented around 250 B.C., for estimating the sum of numbers is embodied in Shachong. (That’s really hard hardware!)

Like us, many others who study technology affect, algorithms are also used in startling and unexpected ways in the 20th century—at least it looks that way to us now. The algorithms we have selected for this issue come from such disparate areas as communications, health care, manufacturing, economics, weather prediction, defense, and fundamental science. Consider, for example, the area of weather forecasting. In one late-night hall session on the Maryland Shore when someone asked, “Who first ate a sandwich?” the answer was “No one.” But the question of speculation about the observed behavior of gas molecules, someone gave what must be the right answer—“A very hungry person.”

The flip side to “necessity is the mother of invention” is “invention creates its own necessity.” Our need for powerful mathematical tools to analyze complex systems and to predict their behavior brings insights that suggest the need, usually much larger, computation to be done. New algorithms are an integral part of our quest for knowledge. We have come to expect a steady supply of them. We’ve become accustomed to gaining the Moore’s Law factor of two every 18 months. In effect, Moore’s Law is a statement of the exponential growth of computation as a function of problem size. Important new algorithms do not come along every 1.5 years, but when they do, they can change the world.

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 new light on some aspect of computing. A colleague recently claimed that he’d done only 15 minutes of productive work in his office over the past year. I asked him what he had done. He responded that while he’d sketched out a fundamental optimization algorithm, he regarded the previous years of work as “just a waste of time” at a cost such that might not have paid off at all.

Researchers have cracked many hard problems since 1 January 2000. One of the most interesting is the search for life in the next century. In spite of a lot of good work, the question of how to extract information from extremely large masses of data is still almost entirely open. Another is the search for faster ways to process more “big data” tasks, too. For example, we need efficient methods to tell when the result of a large number of parallel computations is “good enough.” That is, the way that the search function. The added computational cost is very small, but the added confidence in the answer is large.

Another is the search for better ways to do “meta-computation optimization!” At an even deeper level is the issue of reasonable methods for solving specific cases of “impossible” problems. For example, how do we find the best solution to a problem in attempting to answer many practical questions. Are there efficient ways to attack them?

As we move forward, things will be ripe for another revolution in our understanding of the basis of computation theory. Questions already arising from quantum computing, for example, will force us to reassess the notion of random numbers seem to require that we somehow tie theories of computing, logic, and the nature of the

The new century is going to be very useful for us, but it is not going to be dull either! ■

2 COMPUTING IN SCIENCE & ENGINEERING

Why study algorithms?

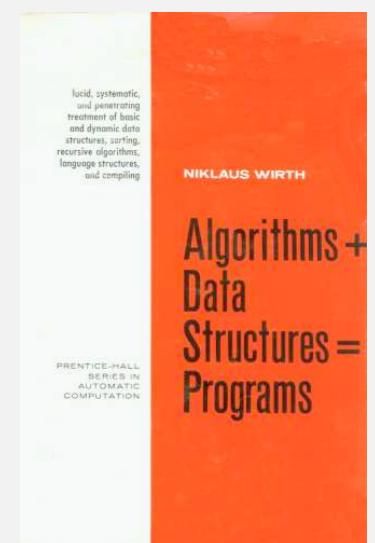
To become a proficient programmer.

“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



Why study algorithms?

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

Computational models are replacing math models in scientific inquiry.

$$E = mc^2$$

$$F = ma$$

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) = E \Psi(r)$$

20th century science
(formula based)

$$F = \frac{Gm_1 m_2}{r^2}$$

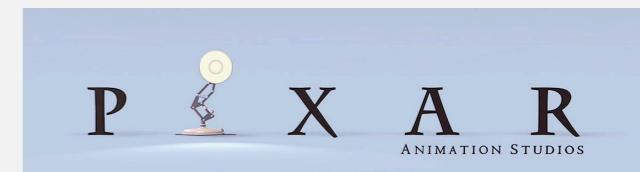
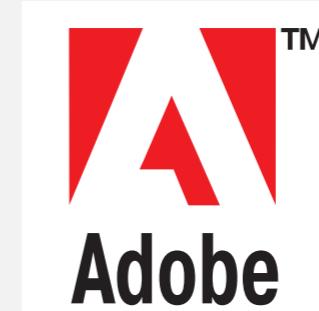
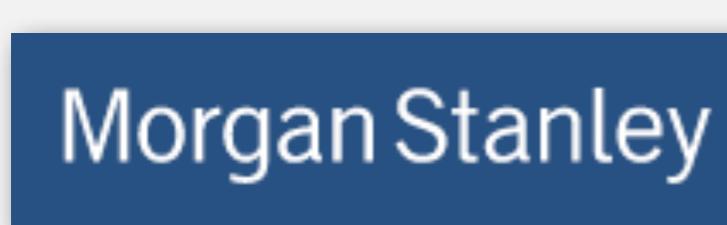
```
for (double t = 0.0; true; t = t + dt)
    for (int i = 0; i < N; i++)
    {
        bodies[i].resetForce();
        for (int j = 0; j < N; j++)
            if (i != j)
                bodies[i].addForce(bodies[j]);
    }
```

21st century science
(algorithm based)

“Algorithms: a common language for nature, human, and computer.” — Avi 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.

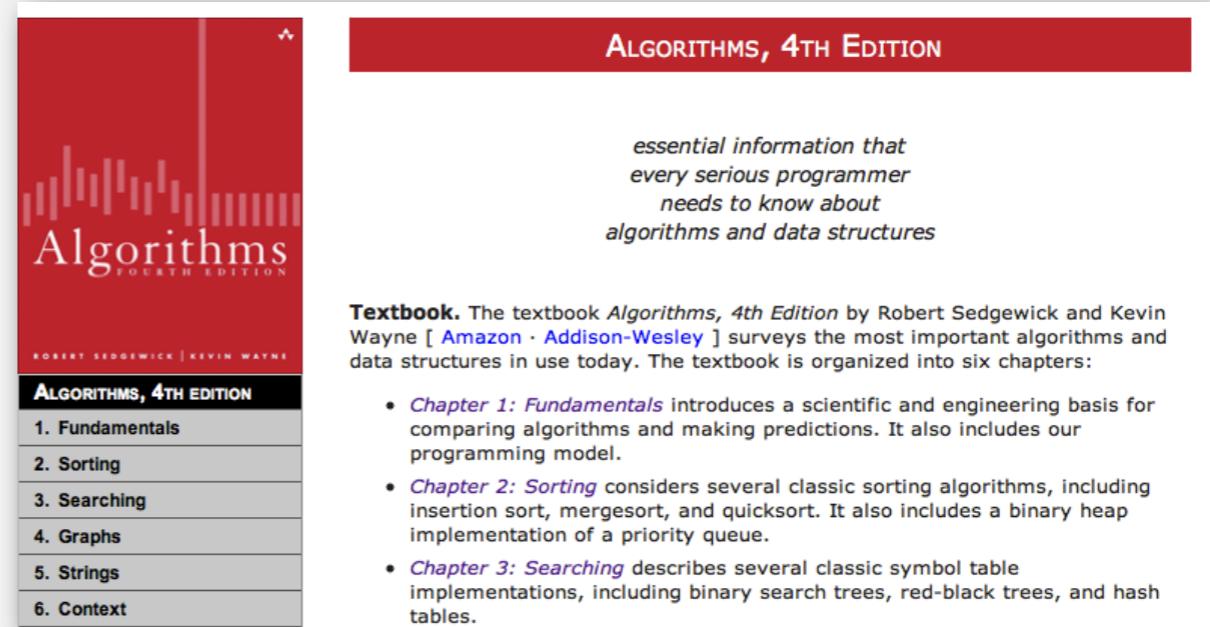
Why study anything else?



Resources

Booksite.

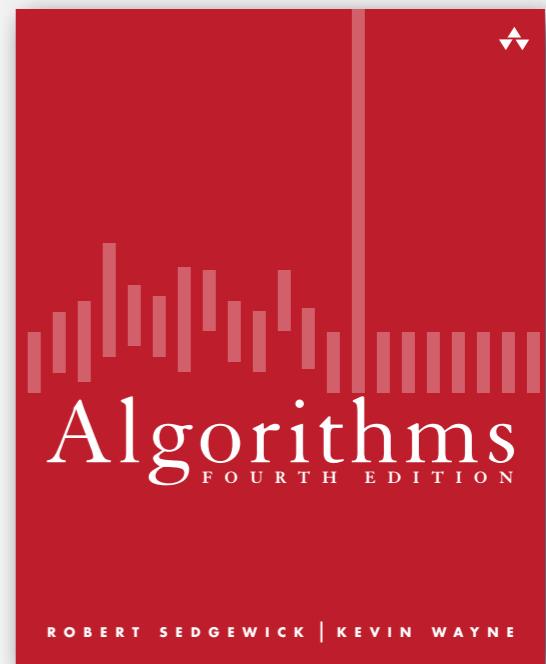
- Lecture slides.
- Download code.
- Summary of content.



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

Textbook (optional).

- *Algorithms, 4th edition* by Sedgewick and Wayne.
- More extensive coverage of topics.
- More topics.



ISBN 0-321-57351-X

Prerequisites

Prerequisites.

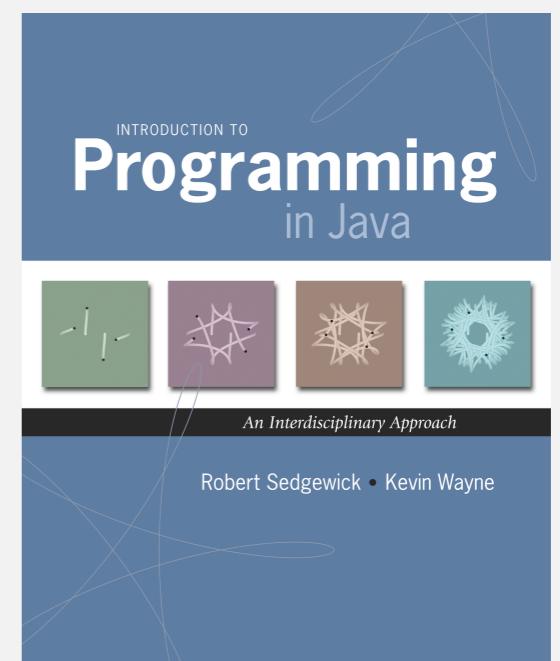
- Programming: loops, arrays, functions, objects, recursion.
- Java: we use as expository language.
- Mathematics: high-school algebra.

Review of prerequisite material.

- Quick: Sections 1.1 and 1.2 of *Algorithms, 4th edition*.
- In-depth: *An Introduction to programming in Java: an interdisciplinary approach* by Sedgewick and Wayne.

Programming environment.

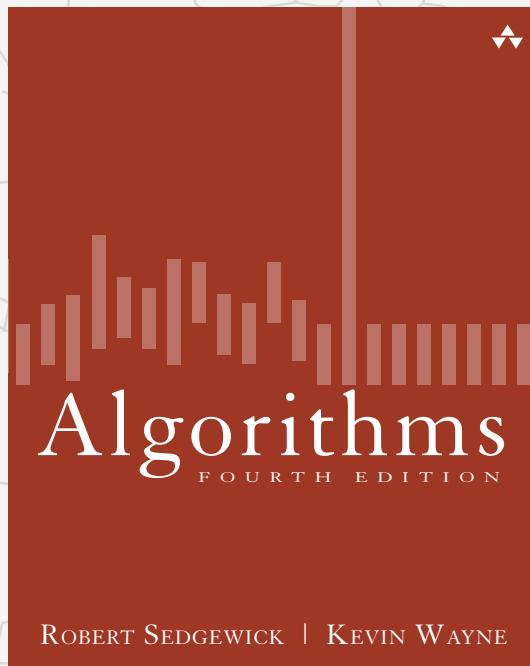
- Use your own, e.g., Eclipse.
- Download ours (see instructions on web).



Quick exercise. Write a Java program.

ISBN 0-321-49805-4

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



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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Cast of characters



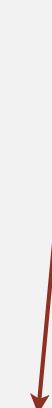
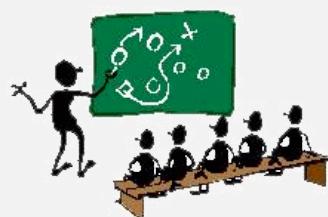
Programmer needs to develop
a working solution.



Client wants to solve
problem efficiently.



Theoretician wants
to understand.

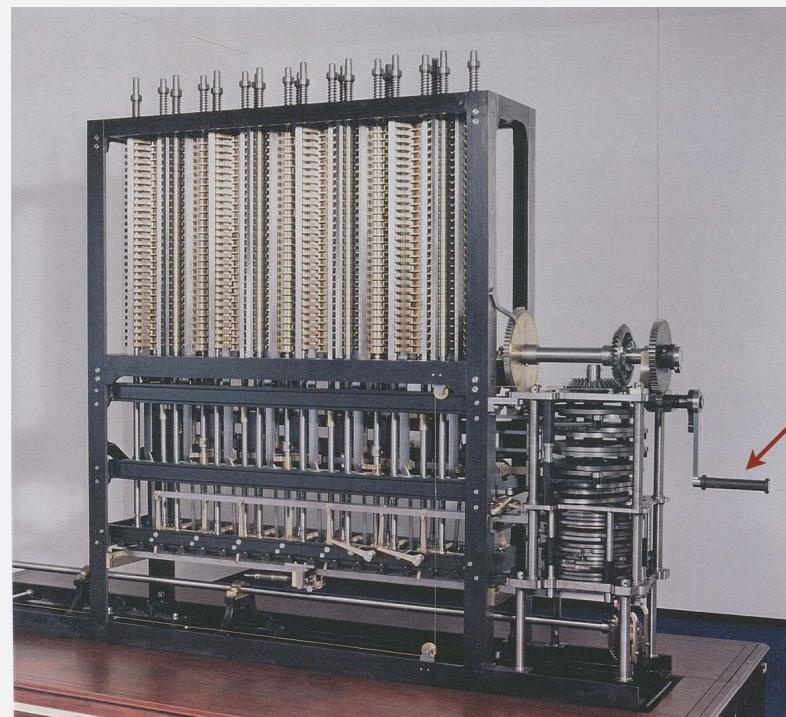
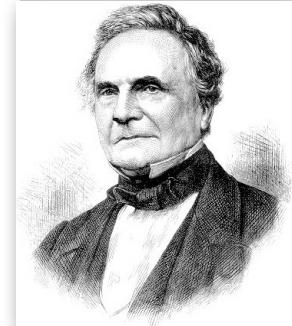


Student might play
any or all of these
roles someday.

Basic **blocking** and **tackling**
is sometimes necessary.
[this lecture]

Running time

“As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ” — Charles Babbage (1864)

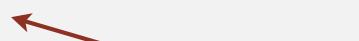


how many times do you have to turn the crank?

Analytic Engine

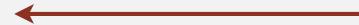
Reasons to analyze algorithms

Predict performance.

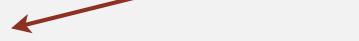


this course

Compare algorithms.



Provide guarantees.



Understand theoretical basis.



theory of algorithms

Primary practical reason: avoid performance bugs.



**client gets poor performance because programmer
did not understand performance characteristics**



Some algorithmic successes

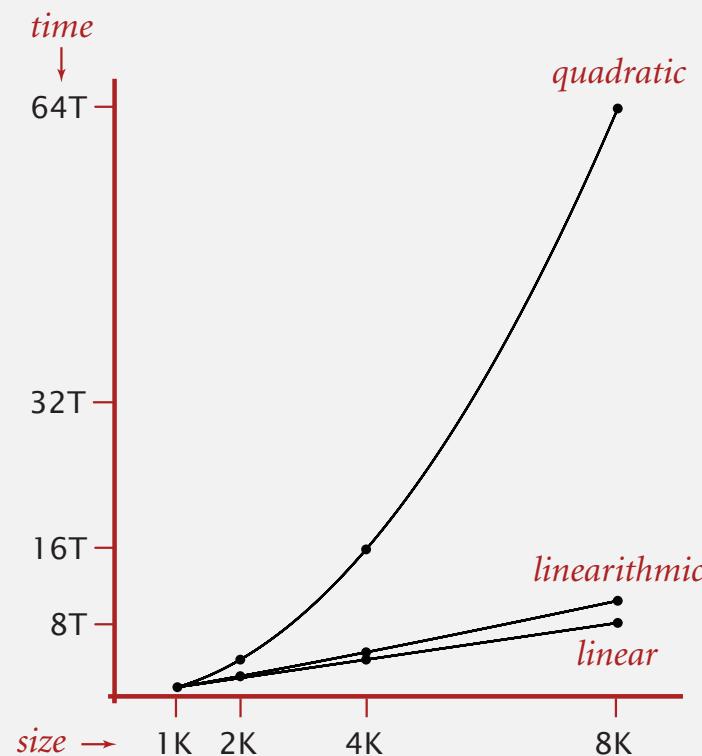
Discrete Fourier transform.

- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics,
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, **enables new technology**.



Friedrich Gauss

1805



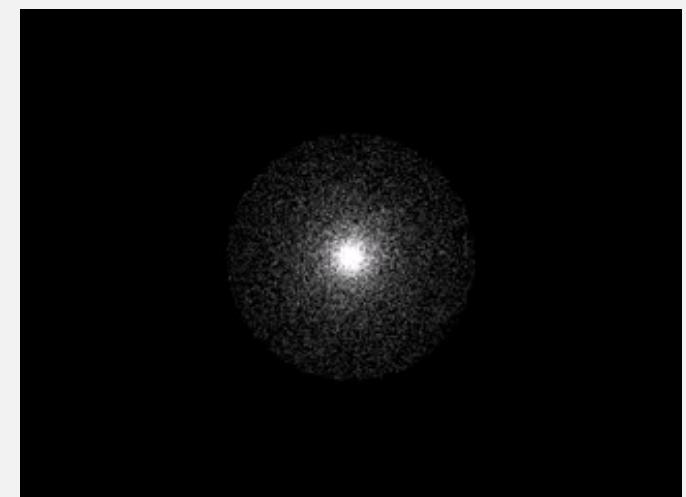
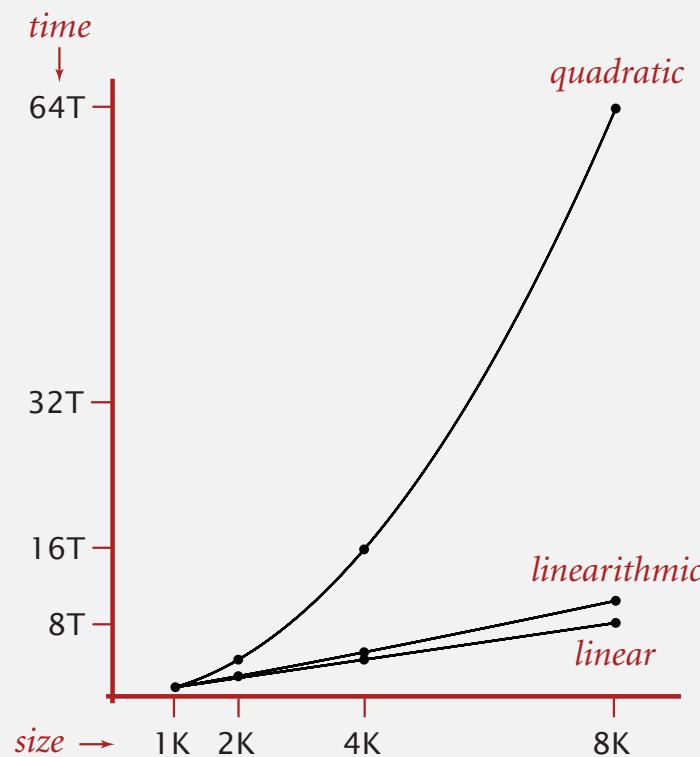
Some algorithmic successes

N-body simulation.

- Simulate gravitational interactions among N bodies.
- Brute force: N^2 steps.
- Barnes-Hut algorithm: $N \log N$ steps, enables new research.



Andrew Appel
PU '81



The challenge

Q. Will my program be able to solve a large practical input?

Why is my program so slow ?

Why does it run out of memory ?



Insight. [Knuth 1970s] Use scientific method to understand performance.

Scientific method applied to analysis of algorithms

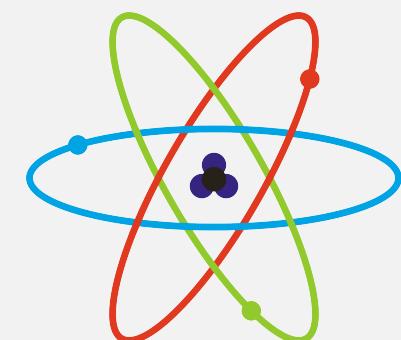
A framework for predicting performance and comparing algorithms.

Scientific method.

- **Observe** some feature of the natural world.
- **Hypothesize** a model that is consistent with the observations.
- **Predict** events using the hypothesis.
- **Verify** the predictions by making further observations.
- **Validate** by repeating until the hypothesis and observations agree.

Principles.

- Experiments must be **reproducible**.
- Hypotheses must be **falsifiable**.



Feature of the natural world. Computer itself.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ ***observations***
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Example: 3-SUM

3-SUM. Given N distinct integers, how many triples sum to exactly zero?

```
% more 8ints.txt  
8  
30 -40 -20 -10 40 0 10 5  
  
% java ThreeSum 8ints.txt  
4
```

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0

Context. Deeply related to problems in computational geometry.

3-SUM: brute-force algorithm

```
public class ThreeSum
{
    public static int count(int[] a)
    {
        int N = a.length;
        int count = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                for (int k = j+1; k < N; k++) ← check each triple
                    if (a[i] + a[j] + a[k] == 0) ← for simplicity, ignore
                        count++;                                integer overflow
        return count;
    }

    public static void main(String[] args)
    {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

Measuring the running time

Q. How to time a program?

A. Manual.

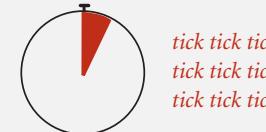


```
% java ThreeSum 1Kints.txt
```



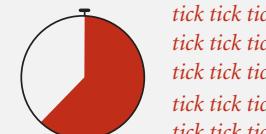
70

```
% java ThreeSum 2Kints.txt
```



528

```
% java ThreeSum 4Kints.txt
```



4039

Measuring the running time

Q. How to time a program?

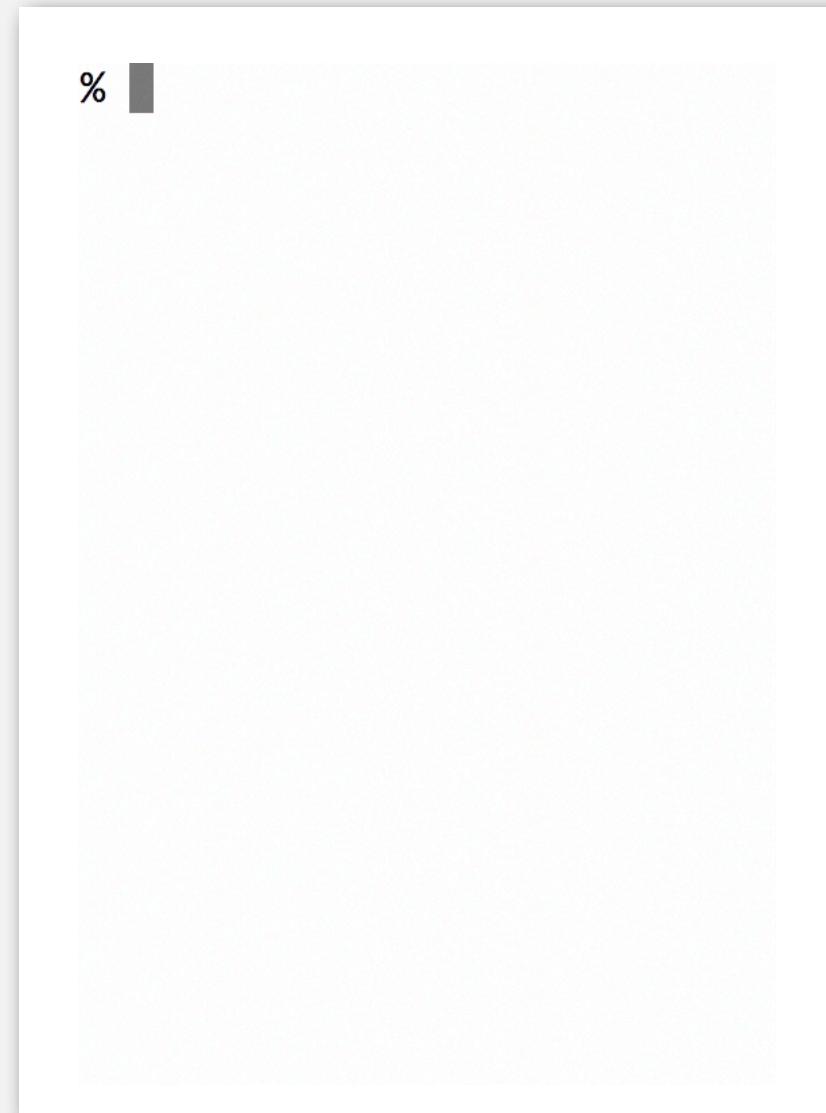
A. Automatic.

```
public class Stopwatch  (part of stdlib.jar )  
  
    Stopwatch()           create a new stopwatch  
  
    double elapsedTime() time since creation (in seconds)
```

```
public static void main(String[] args)  
{  
    int[] a = In.readInts(args[0]);  
    Stopwatch stopwatch = new Stopwatch();  
    StdOut.println(ThreeSum.count(a));  
    double time = stopwatch.elapsedTime();  
}
```

Empirical analysis

Run the program for various input sizes and measure running time.



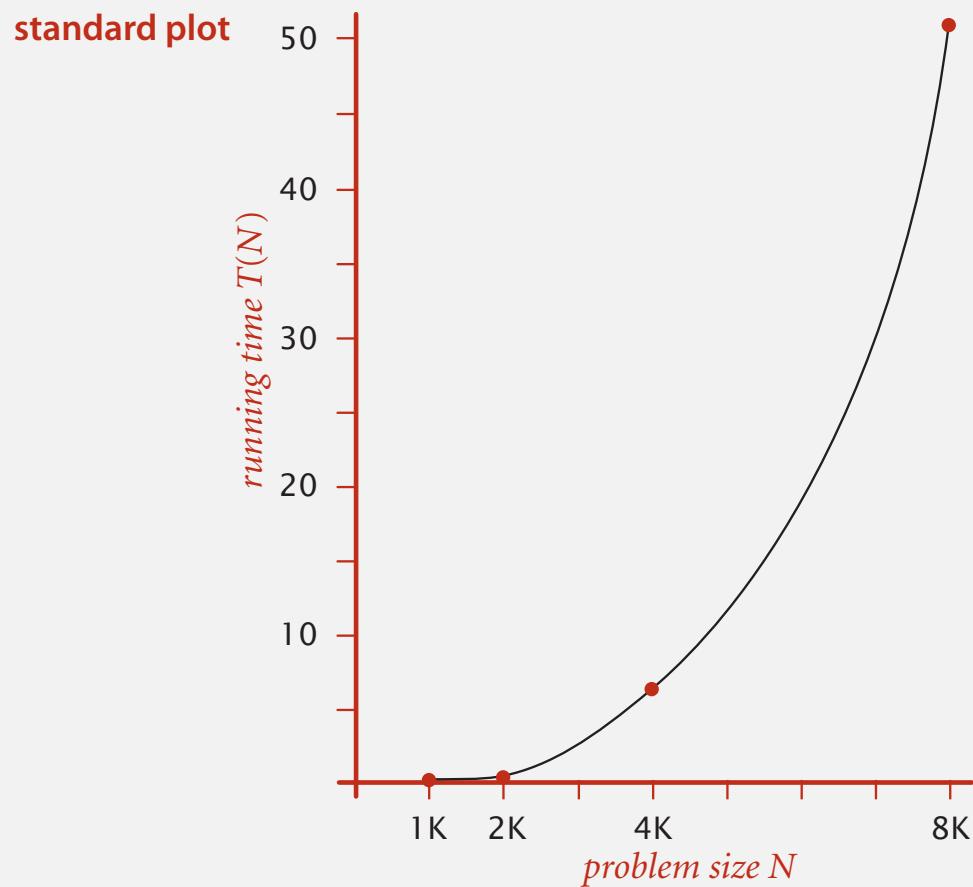
Empirical analysis

Run the program for various input sizes and measure running time.

N	time (seconds) †
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

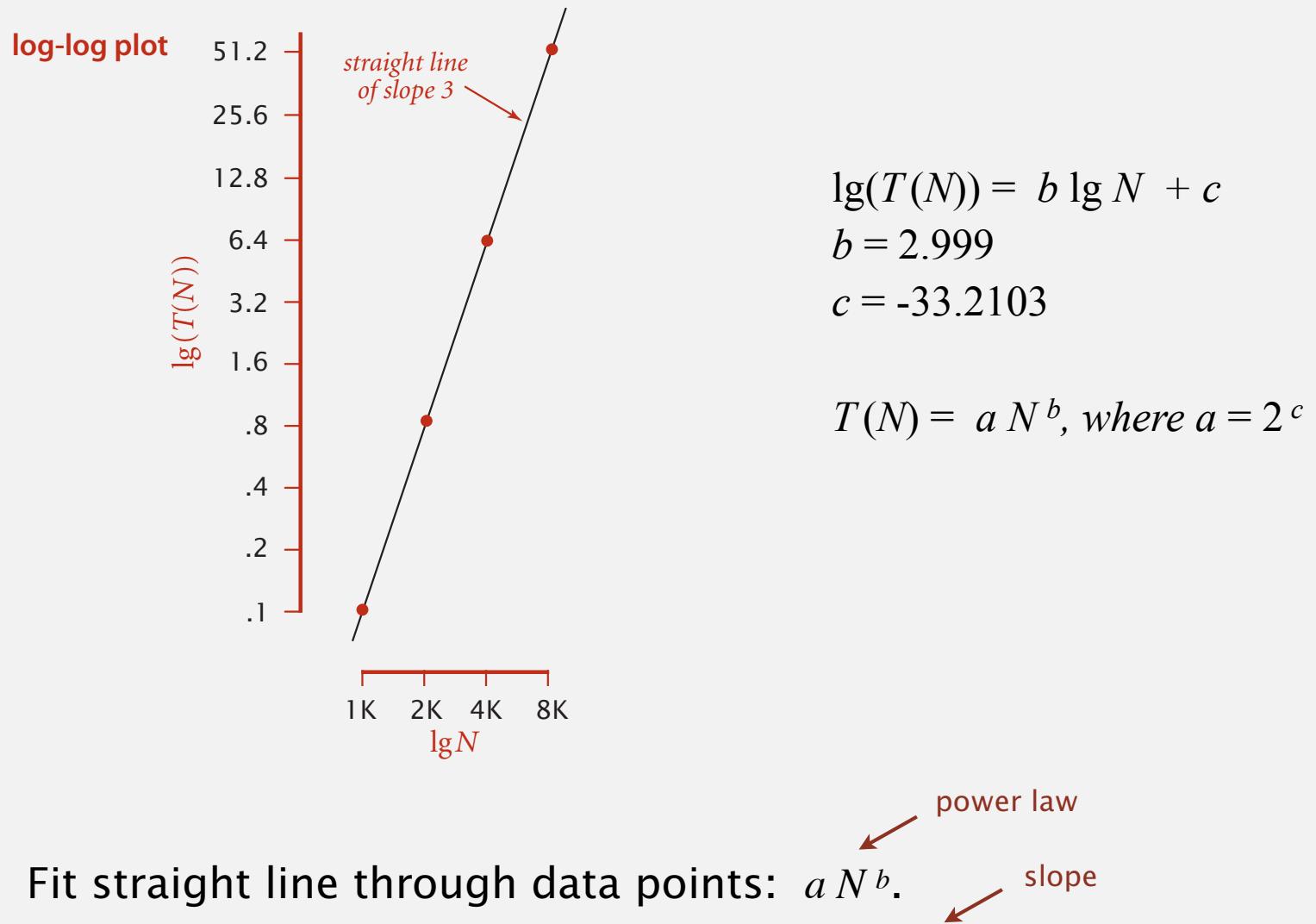
Data analysis

Standard plot. Plot running time $T(N)$ vs. input size N .



Data analysis

Log-log plot. Plot running time $T(N)$ vs. input size N using log-log scale.



Regression. Fit straight line through data points: $a N^b$.

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.



"order of growth" of running time is about N^3 [stay tuned]

Predictions.

- 51.0 seconds for $N = 8,000$.
- 408.1 seconds for $N = 16,000$.

Observations.

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Run program, **doubling** the size of the input.

N	time (seconds) [†]	ratio	lg ratio
250	0.0		—
500	0.0	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8.0	3.0
8,000	51.1	8.0	3.0

↑
seems to converge to a constant $b \approx 3$

Hypothesis. Running time is about $a N^b$ with $b = \lg$ ratio.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Q. How to estimate a (assuming we know b) ?

A. Run the program (for a sufficient large value of N) and solve for a .

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1

$$51.1 = a \times 8000^3$$

$$\Rightarrow a = 0.998 \times 10^{-10}$$

Hypothesis. Running time is about $0.998 \times 10^{-10} \times N^3$ seconds.



almost identical hypothesis
to one obtained via linear regression

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent b
in power law

System dependent effects.

- Hardware: CPU, memory, cache, ...
- Software: compiler, interpreter, garbage collector, ...
- System: operating system, network, other apps, ...

determines constant a
in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.



e.g., can run huge number of experiments

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ ***observations***
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

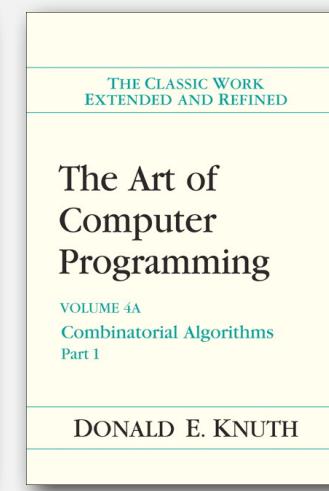
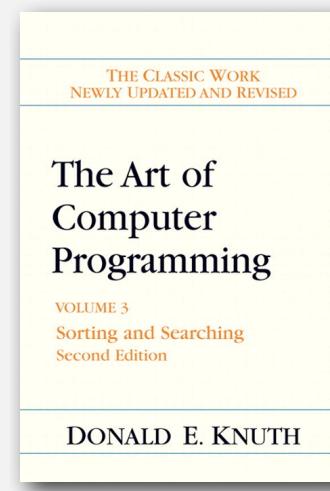
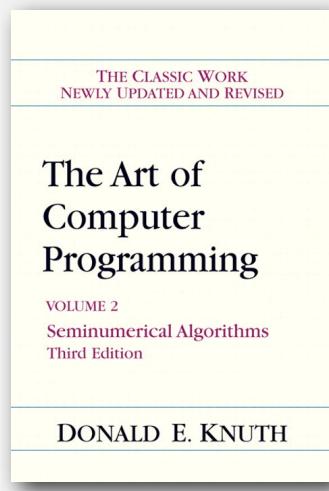
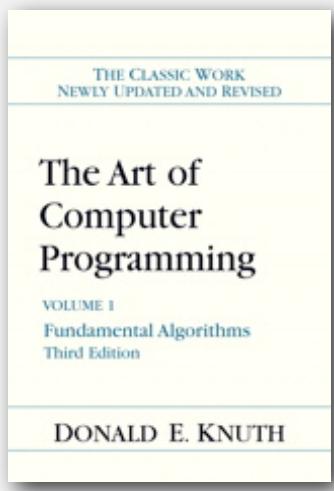
1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ ***mathematical models***
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Mathematical models for running time

Total running time: sum of cost \times frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

operation	example	nanoseconds †
integer add	$a + b$	2.1
integer multiply	$a * b$	2.4
integer divide	a / b	5.4
floating-point add	$a + b$	4.6
floating-point multiply	$a * b$	4.2
floating-point divide	a / b	13.5
sine	<code>Math.sin(theta)</code>	91.3
arctangent	<code>Math.atan2(y, x)</code>	129.0
...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

operation	example	nanoseconds †
variable declaration	<code>int a</code>	c_1
assignment statement	<code>a = b</code>	c_2
integer compare	<code>a < b</code>	c_3
array element access	<code>a[i]</code>	c_4
array length	<code>a.length</code>	c_5
1D array allocation	<code>new int[N]</code>	$c_6 N$
2D array allocation	<code>new int[N][N]</code>	$c_7 N^2$
string length	<code>s.length()</code>	c_8
substring extraction	<code>s.substring(N/2, N)</code>	c_9
string concatenation	<code>s + t</code>	$c_{10} N$

Novice mistake. Abusive string concatenation.

Example: 1-SUM

Q. How many instructions as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    if (a[i] == 0)  
        count++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2N$

Example: 2-SUM

Q. How many instructions as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

$$\begin{aligned}0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2}N(N - 1) \\&= \binom{N}{2}\end{aligned}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$
equal to compare	$\frac{1}{2}N(N - 1)$
array access	$N(N - 1)$
increment	$\frac{1}{2}N(N - 1)$ to $N(N - 1)$

tedious to count exactly

Simplifying the calculations

“It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. ” — Alan Turing

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

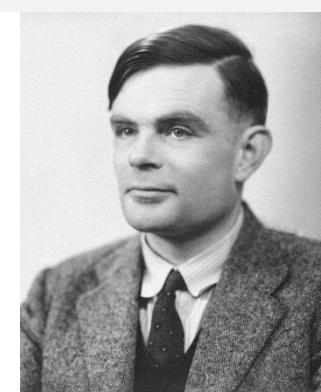
By A. M. TURING

(National Physical Laboratory, Teddington, Middlesex)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known ‘Gauss elimination process’, it is found that the errors are normally quite moderate: no exponential build-up need occur.



Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

$$\begin{aligned}0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2}N(N - 1) \\&= \binom{N}{2}\end{aligned}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$
equal to compare	$\frac{1}{2}N(N - 1)$
array access	$N(N - 1)$
increment	$\frac{1}{2}N(N - 1)$ to $N(N - 1)$

cost model = array accesses
(we assume compiler/JVM do not optimize array accesses away!)

Simplification 2: tilde notation

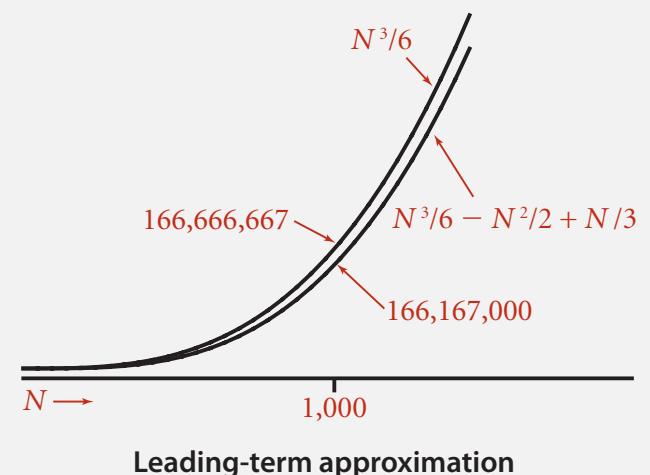
- Estimate running time (or memory) as a function of input size N .
- Ignore lower order terms.
 - when N is large, terms are negligible
 - when N is small, we don't care

Ex 1. $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$

Ex 2. $\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$

Ex 3. $\frac{1}{6}N^3 - \frac{1}{2}N^2 + \frac{1}{3}N \sim \frac{1}{6}N^3$

discard lower-order terms
(e.g., $N = 1000$: 500 thousand vs. 166 million)



Technical definition. $f(N) \sim g(N)$ means $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1$

Simplification 2: tilde notation

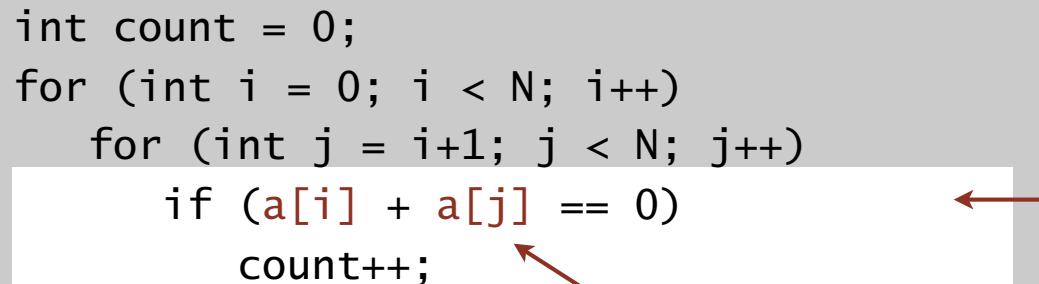
- Estimate running time (or memory) as a function of input size N .
- Ignore lower order terms.
 - when N is large, terms are negligible
 - when N is small, we don't care

operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N (N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$\frac{1}{2} N (N - 1)$ to $N (N - 1)$	$\sim \frac{1}{2} N^2$ to $\sim N^2$

Example: 2-SUM

Q. Approximately how many array accesses as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```



A. $\sim N^2$ array accesses.

$$\begin{aligned}0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2} N(N - 1) \\&= \binom{N}{2}\end{aligned}$$

Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size N ?

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0) ← "inner loop"
                count++;
```

A. $\sim \frac{1}{2} N^3$ array accesses.

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!}$$
$$\sim \frac{1}{6} N^3$$

Bottom line. Use cost model and tilde notation to simplify counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take discrete mathematics course.

A2. Replace the sum with an integral, and use calculus!

Ex 1. $1 + 2 + \dots + N$.

$$\sum_{i=1}^N i \sim \int_{x=1}^N x dx \sim \frac{1}{2} N^2$$

Ex 2. $1^k + 2^k + \dots + N^k$.

$$\sum_{i=1}^N i^k \sim \int_{x=1}^N x^k dx \sim \frac{1}{k+1} N^{k+1}$$

Ex 3. $1 + 1/2 + 1/3 + \dots + 1/N$.

$$\sum_{i=1}^N \frac{1}{i} \sim \int_{x=1}^N \frac{1}{x} dx = \ln N$$

Ex 4. 3-sum triple loop.

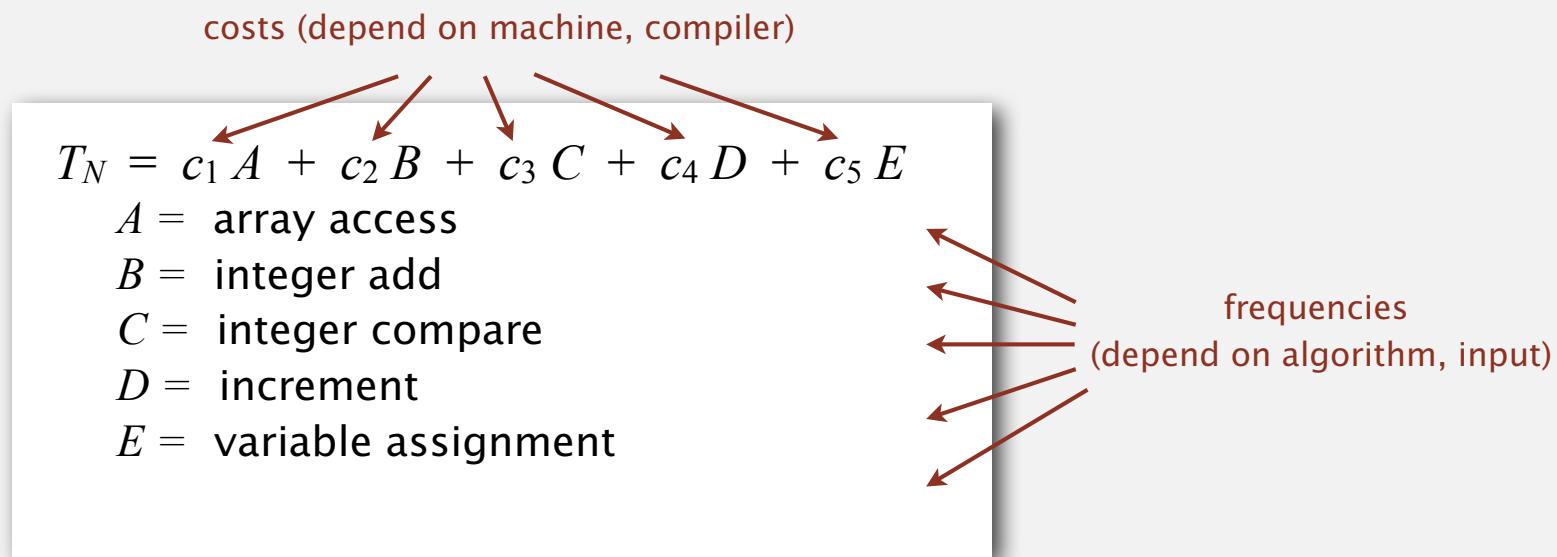
$$\sum_{i=1}^N \sum_{j=i}^N \sum_{k=j}^N 1 \sim \int_{x=1}^N \int_{y=x}^N \int_{z=y}^N dz dy dx \sim \frac{1}{6} N^3$$

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



Bottom line. We use approximate models in this course: $T(N) \sim c N^3$.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ ***mathematical models***
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ ***order-of-growth classifications***
- ▶ *theory of algorithms*
- ▶ *memory*

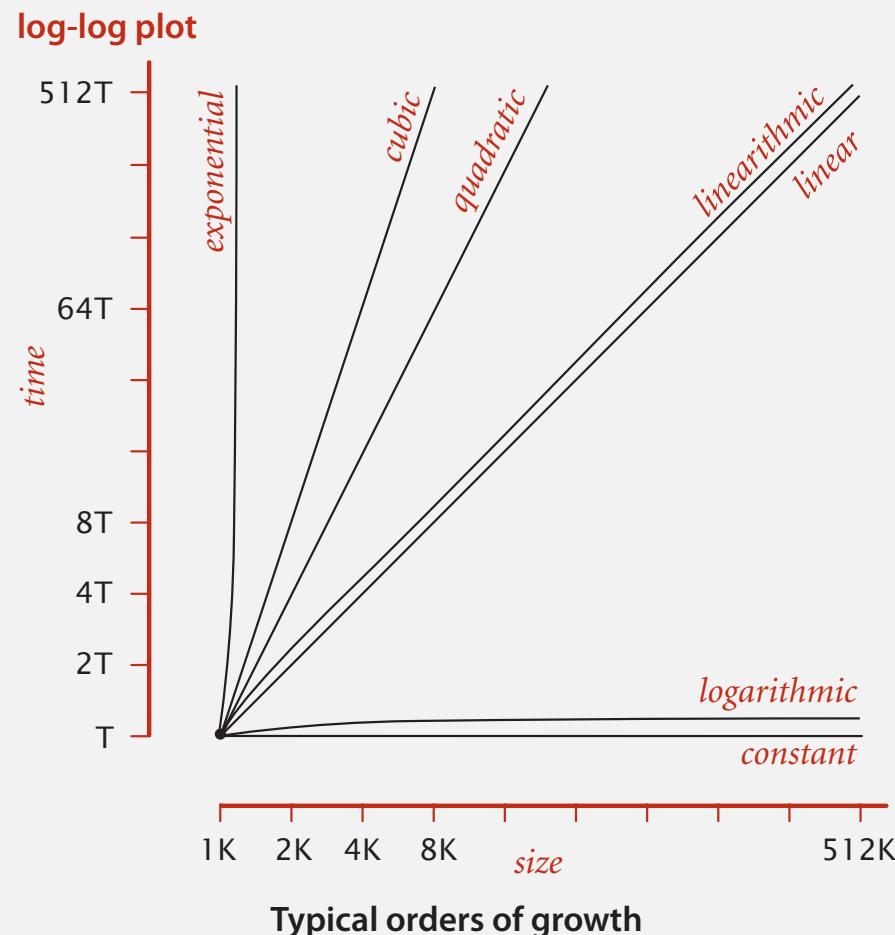
Common order-of-growth classifications

Good news. the small set of functions

$1, \log N, N, N \log N, N^2, N^3$, and 2^N

suffices to describe order-of-growth of typical algorithms.

order of growth discards
leading coefficient



Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	$a = b + c;$	statement	add two numbers	1
$\log N$	logarithmic	<pre>while (N > 1) { N = N / 2; ... }</pre>	divide in half	binary search	~ 1
N	linear	<pre>for (int i = 0; i < N; i++) { ... }</pre>	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</pre>	double loop	check all pairs	4
N^3	cubic	<pre>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }</pre>	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
$\log N$	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
$N \log N$	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Bottom line. Need linear or linearithmic alg to keep pace with Moore's law.

Binary search demo

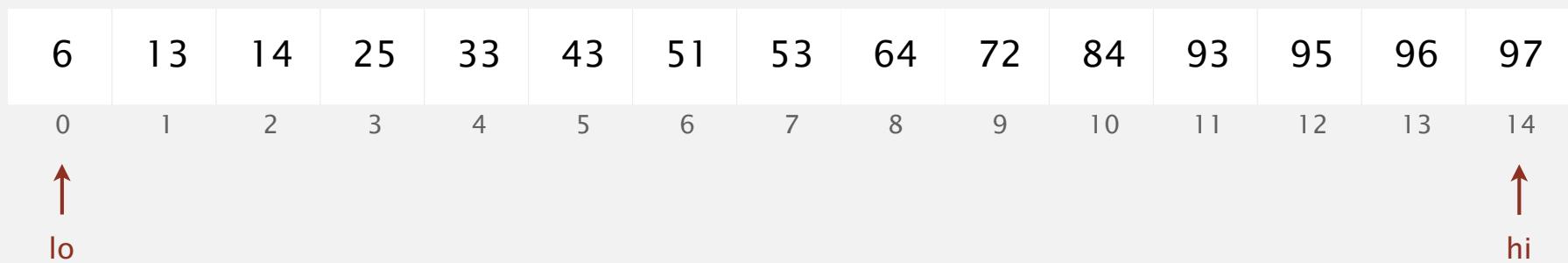
Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



successful search for 33



Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946; first bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

one "3-way compare"

Invariant. If key appears in the array `a[]`, then $a[lo] \leq key \leq a[hi]$.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

↑
left or right half

↑
possible to implement with one
2-way compare (instead of 3-way)

Pf sketch.

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && \text{given} \\ &\leq T(N/4) + 1 + 1 && \text{apply recurrence to first term} \\ &\leq T(N/8) + 1 + 1 + 1 && \text{apply recurrence to first term} \\ &\dots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && \text{stop applying, } T(1) = 1 \\ &= 1 + \lg N \end{aligned}$$

An $N^2 \log N$ algorithm for 3-SUM

Sorting-based algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers $a[i]$ and $a[j]$, binary search for $-(a[i] + a[j])$.

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10 0 5 10 30 40

binary search

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-40, 40)	0
⋮	⋮
(-20, -10)	30
⋮	⋮
(-10, 0)	10
⋮	⋮
(10, 30)	-40
(10, 40)	-50
(30, 40)	-70

only count if
 $a[i] < a[j] < a[k]$
to avoid
double counting

Analysis. Order of growth is $N^2 \log N$.

- Step 1: N^2 with insertion sort.
- Step 2: $N^2 \log N$ with binary search.

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth \Rightarrow faster in practice.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ ***order-of-growth classifications***
- ▶ *theory of algorithms*
- ▶ *memory*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ ***theory of algorithms***
- ▶ *memory*

Types of analyses

Best case. Lower bound on cost.

- Determined by “easiest” input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

- Need a model for “random” input.
- Provides a way to predict performance.

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Comparisons for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. “Expected” cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

Theory of algorithms

Goals.

- Establish “difficulty” of a problem.
- Develop “optimal” algorithms.

Approach.

- Suppress details in analysis: analyze “to within a constant factor”.
- Eliminate variability in input model by focusing on the worst case.

Optimal algorithm.

- Performance guarantee (to within a constant factor) for any input.
- No algorithm can provide a better performance guarantee.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ 10 N^2 $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	10 N^2 100 N $22 N \log N + 3 N$ ⋮	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ ⋮	develop lower bounds

Theory of algorithms: example 1

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = “*Is there a 0 in the array?*”

Upper bound.

A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound.

Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

Upper bound.

A specific algorithm.

- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound.

Proof that no algorithm can do better.

- Ex. Have to examine all N entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s-.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than “to within a constant factor” to predict performance.

Commonly-used notations

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10 N^2$	$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.

This course. Focus on approximate models: use Tilde-notation

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ ***theory of algorithms***
- ▶ *memory*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ ***memory***

Basics

Bit. 0 or 1. NIST most computer scientists
Byte. 8 bits. ↓ ↓
Megabyte (MB). 1 million or 2^{20} bytes.
Gigabyte (GB). 1 billion or 2^{30} bytes.



64-bit machine. We assume a 64-bit machine with 8 byte pointers.

- Can address more memory.
- Pointers use more space.

some JVMs "compress" ordinary object
pointers to 4 bytes to avoid this cost



Typical memory usage for primitive types and arrays

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

for primitive types

type	bytes
char[]	$2N + 24$
int[]	$4N + 24$
double[]	$8N + 24$

for one-dimensional arrays

type	bytes
char[][]	$\sim 2 MN$
int[][]	$\sim 4 MN$
double[][]	$\sim 8 MN$

for two-dimensional arrays

Typical memory usage for objects in Java

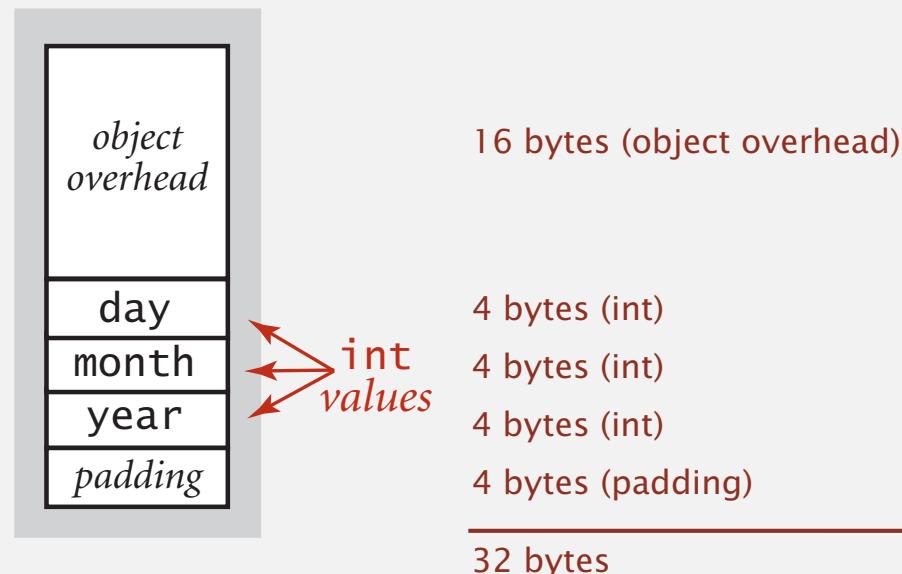
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 1. A Date object uses 32 bytes of memory.

```
public class Date
{
    private int day;
    private int month;
    private int year;
    ...
}
```



Typical memory usage for objects in Java

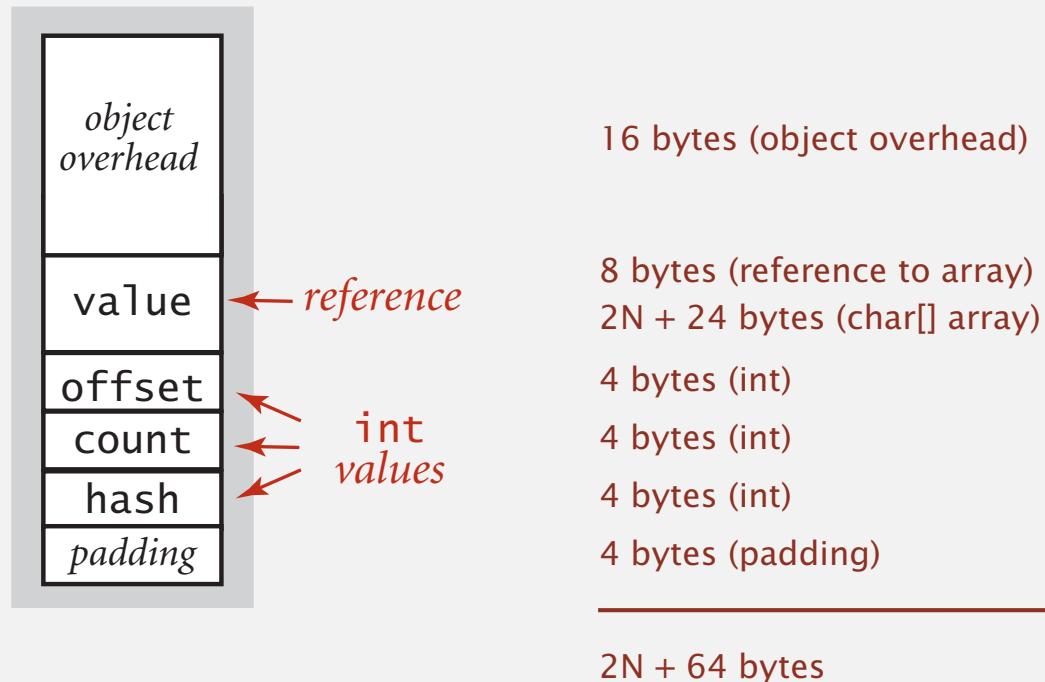
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 2. A virgin String of length N uses $\sim 2N$ bytes of memory.

```
public class String
{
    private char[] value;
    private int offset;
    private int count;
    private int hash;
    ...
}
```



Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for int, 8 bytes for double, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable
+ 8 bytes if inner class (for pointer to enclosing class).
- Padding: round up to multiple of 8 bytes.

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference,
add memory (recursively) for referenced object.

Example

- Q. How much memory does WeightedQuickUnionUF use as a function of N ?
Use tilde notation to simplify your answer.

```
public class WeightedQuickUnionUF
{
```

```
    private int[] id;
    private int[] sz;
    private int count;
```

```
    public WeightedQuickUnionUF(int N)
    {
```

```
        id = new int[N];
        sz = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
        for (int i = 0; i < N; i++) sz[i] = 1;
    }
    ...
}
```

16 bytes
(object overhead)

8 + (4N + 24) each
reference + int[] array

4 bytes (int)

4 bytes (padding)

- A. $8N + 88 \sim 8N$ bytes.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ ***memory***

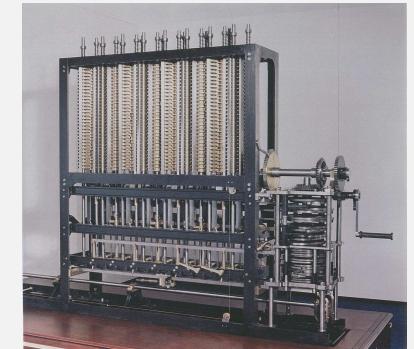
Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and formulate a hypothesis for running time.
- Model enables us to **make predictions**.

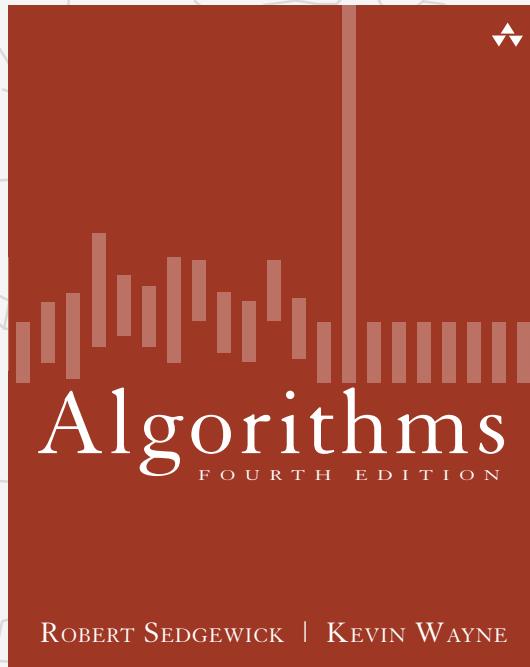
Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to **explain behavior**.



Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.



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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<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.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

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) ✓

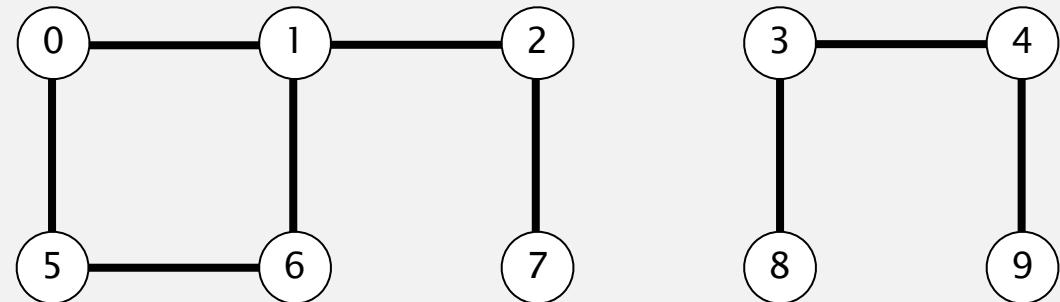
union(5, 0)

union(7, 2)

union(6, 1)

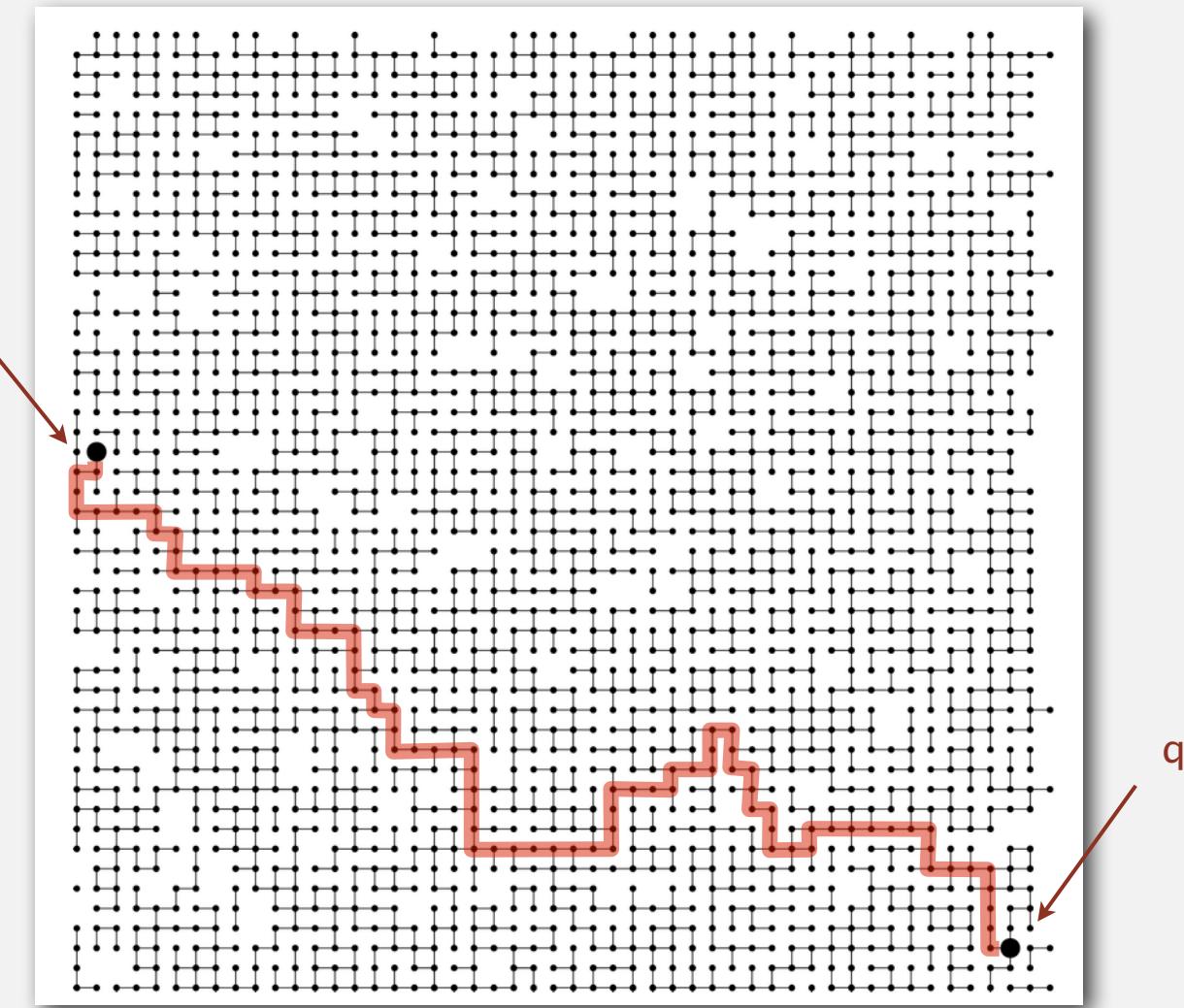
union(1, 0)

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.



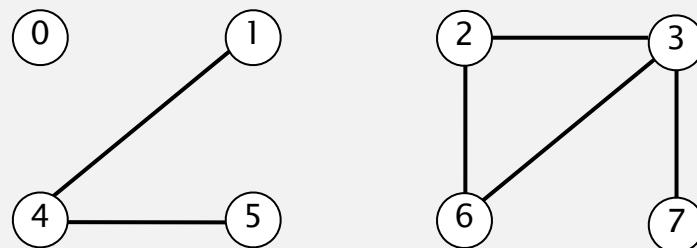
can use symbol table to translate from site
names to integers: stay tuned (Chapter 3)

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.



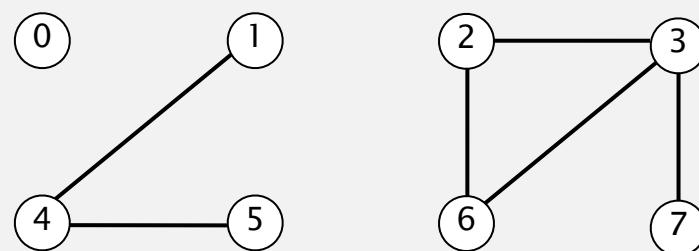
$\{ 0 \} \{ 1 4 5 \} \{ 2 3 6 7 \}$

3 connected components

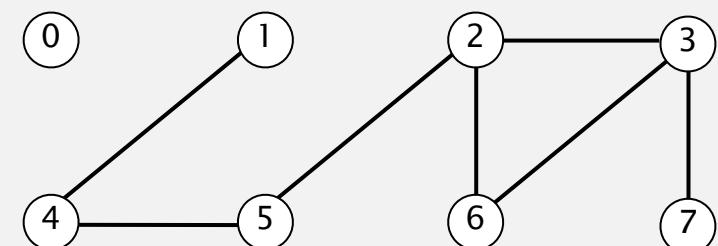
Implementing the operations

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

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



union(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
 - 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
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Quick-find [eager approach]

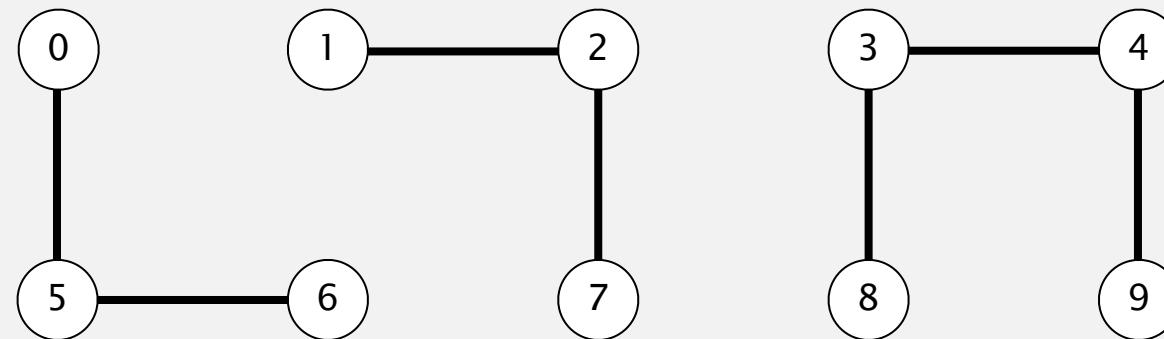
Data structure.

- 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 $\text{id}[]$ 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
$\text{id}[]$	0	1	1	8	8	0	0	1	8	8

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

$\text{id}[6] = 0; \text{id}[1] = 1$
6 and 1 are not connected

Union. To merge components containing p and q , change all entries whose id equals $\text{id}[p]$ to $\text{id}[q]$.

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	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

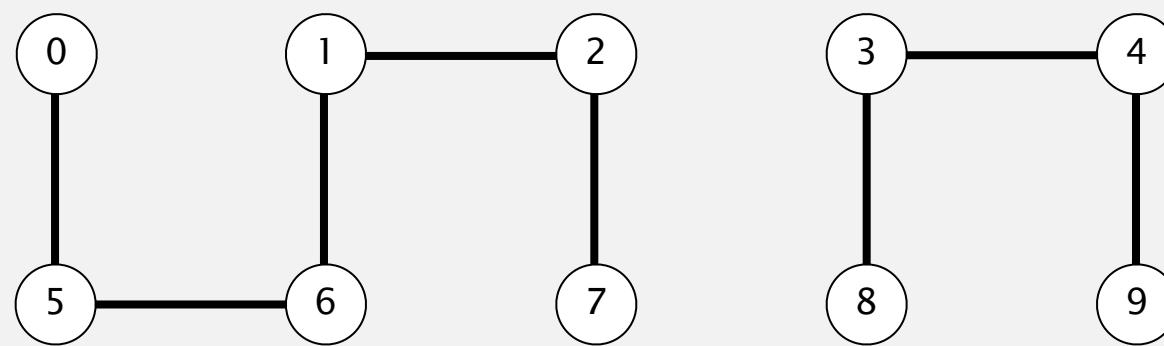
7

8

9

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

Quick-find demo



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

    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 $\text{id}[p]$ to $\text{id}[q]$
(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

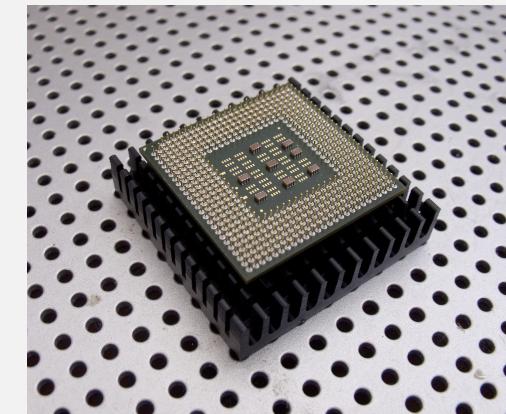


Quadratic algorithms do not scale

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!

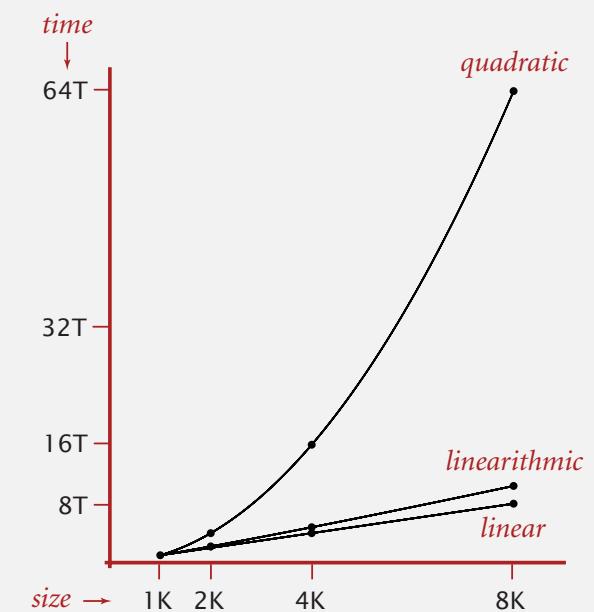


Ex. Huge problem for quick-find.

- 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 ⇒ want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

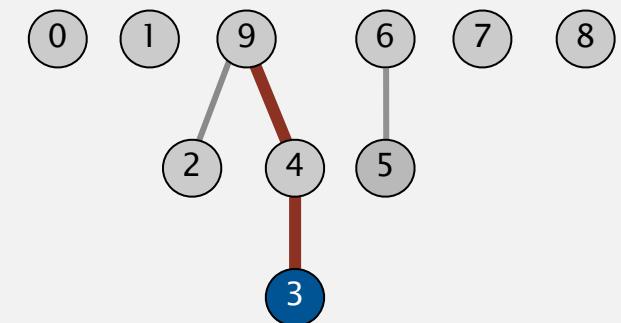
Quick-union [lazy approach]

algorithm design where we try to avoid doing work until we have to

Data structure.

- Integer array $\text{id}[]$ of length N .
- Interpretation: $\text{id}[i]$ is parent of i .
keep going until it doesn't change
(algorithm ensures no cycles)
- Root of i is $\text{id}[\text{id}[\text{id}[\dots\text{id}[i]\dots]]]$.

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	0	1	9	4	9	6	6	7	8	9

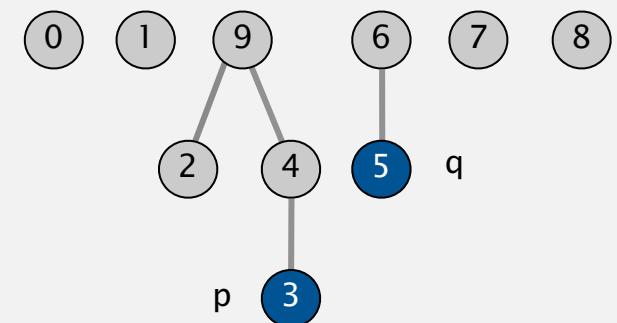


Quick-union [lazy approach]

Data structure.

- Integer array $\text{id}[]$ of length N .
- Interpretation: $\text{id}[i]$ is parent of i .
- Root of i is $\text{id}[\text{id}[\text{id}[\dots\text{id}[i]\dots]]]$.

0	1	2	3	4	5	6	7	8	9	
$\text{id}[]$	0	1	9	4	9	6	6	7	8	9



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

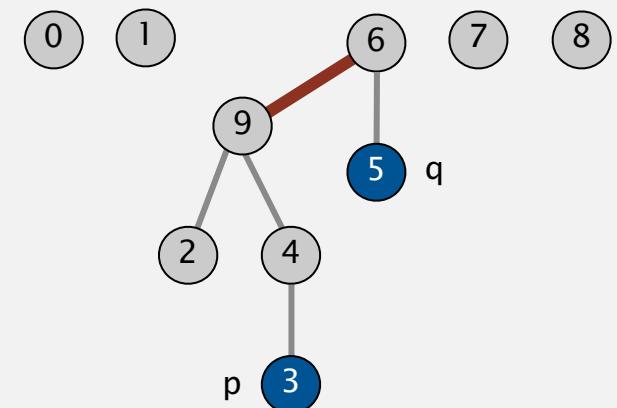
root of 3 is 9
root of 5 is 6

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	
$\text{id}[]$	0	1	9	4	9	6	6	7	8	6

↑
only one value changes



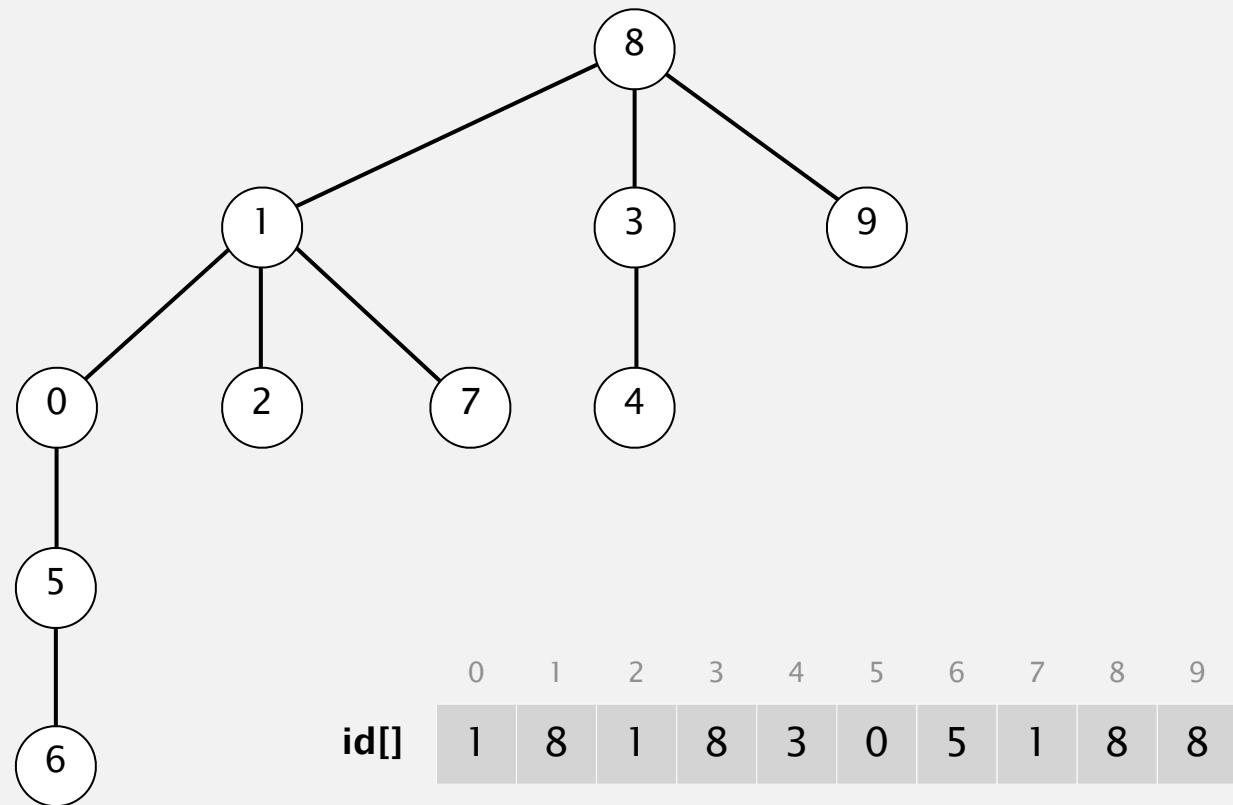
Quick-union demo



0 1 2 3 4 5 6 7 8 9

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

Quick-union demo



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	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 them flat.

Quick-union defect.

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

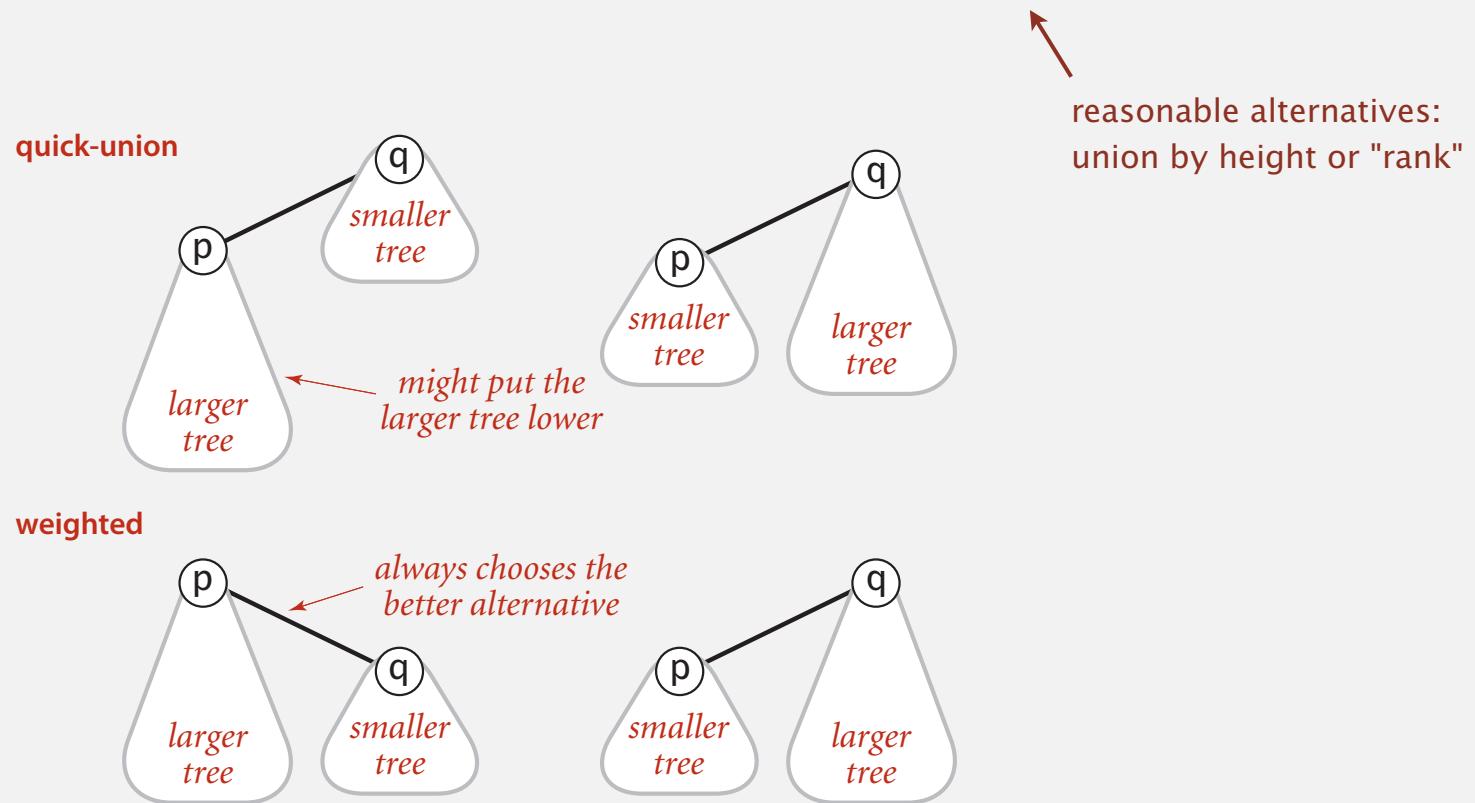
1.5 UNION-FIND

- ▶ *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.



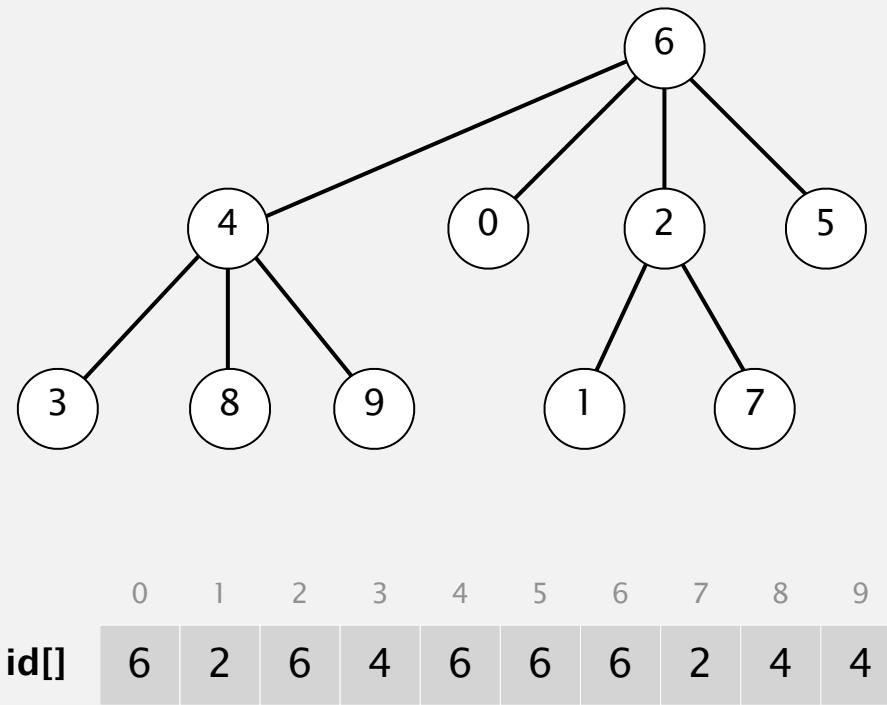
Weighted quick-union demo



0 1 2 3 4 5 6 7 8 9

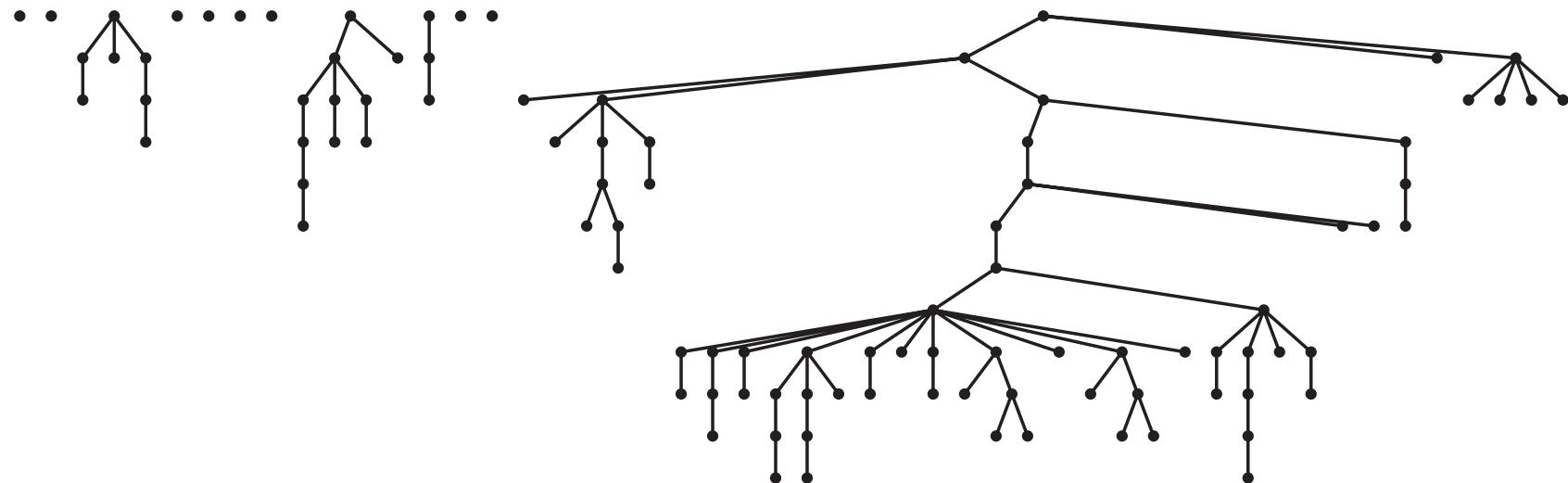
	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

Weighted quick-union demo



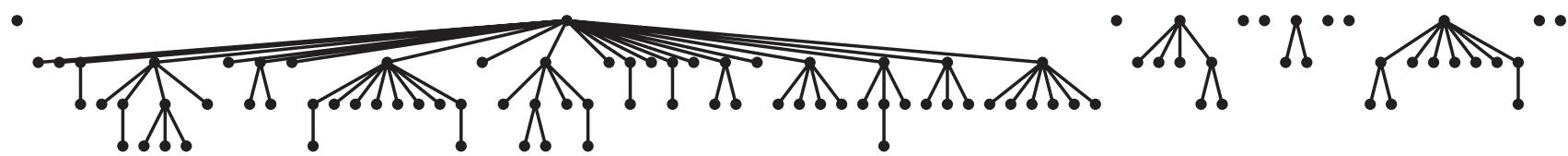
Quick-union and weighted quick-union example

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]; }
```

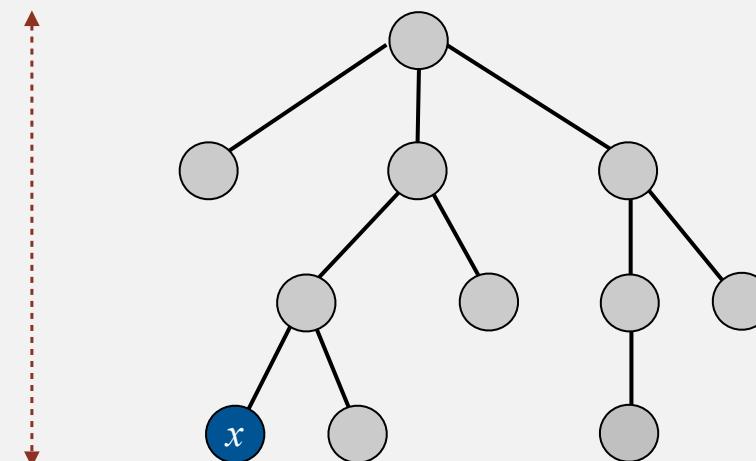
Weighted quick-union analysis

Running time.

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

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

\lg = base-2 logarithm



$$\begin{aligned}N &= 10 \\ \text{depth}(x) &= 3 \leq \lg N\end{aligned}$$

Weighted quick-union analysis

Running time.

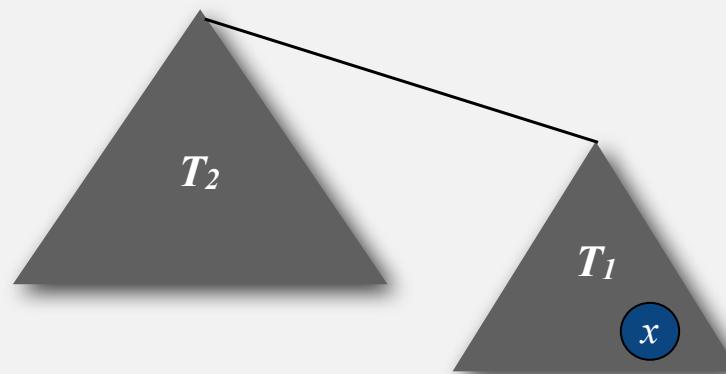
- Find: takes time proportional to depth of p and q .
- Union: takes constant time, given roots.

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 $\lg N$.

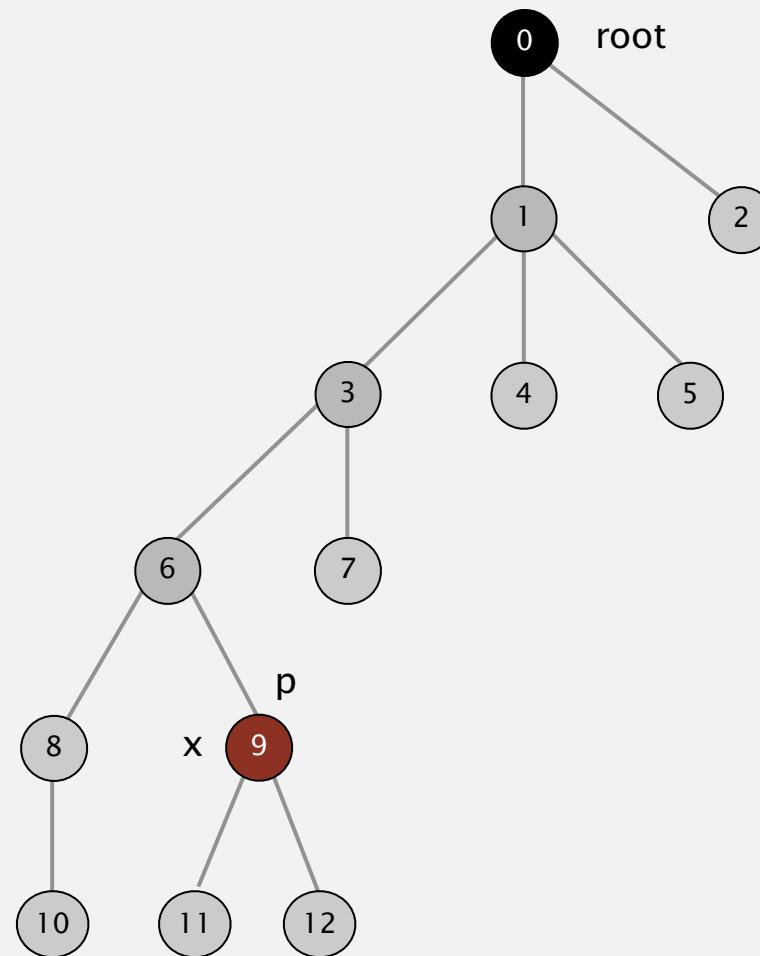
algorithm	initialize	union	connected
quick-find	N	N	1
quick-union	N	N^{\dagger}	N
weighted QU	N	$\lg N^{\dagger}$	$\lg N$

\dagger includes cost of finding roots

- Q. Stop at guaranteed acceptable performance?
A. No, easy to improve further.

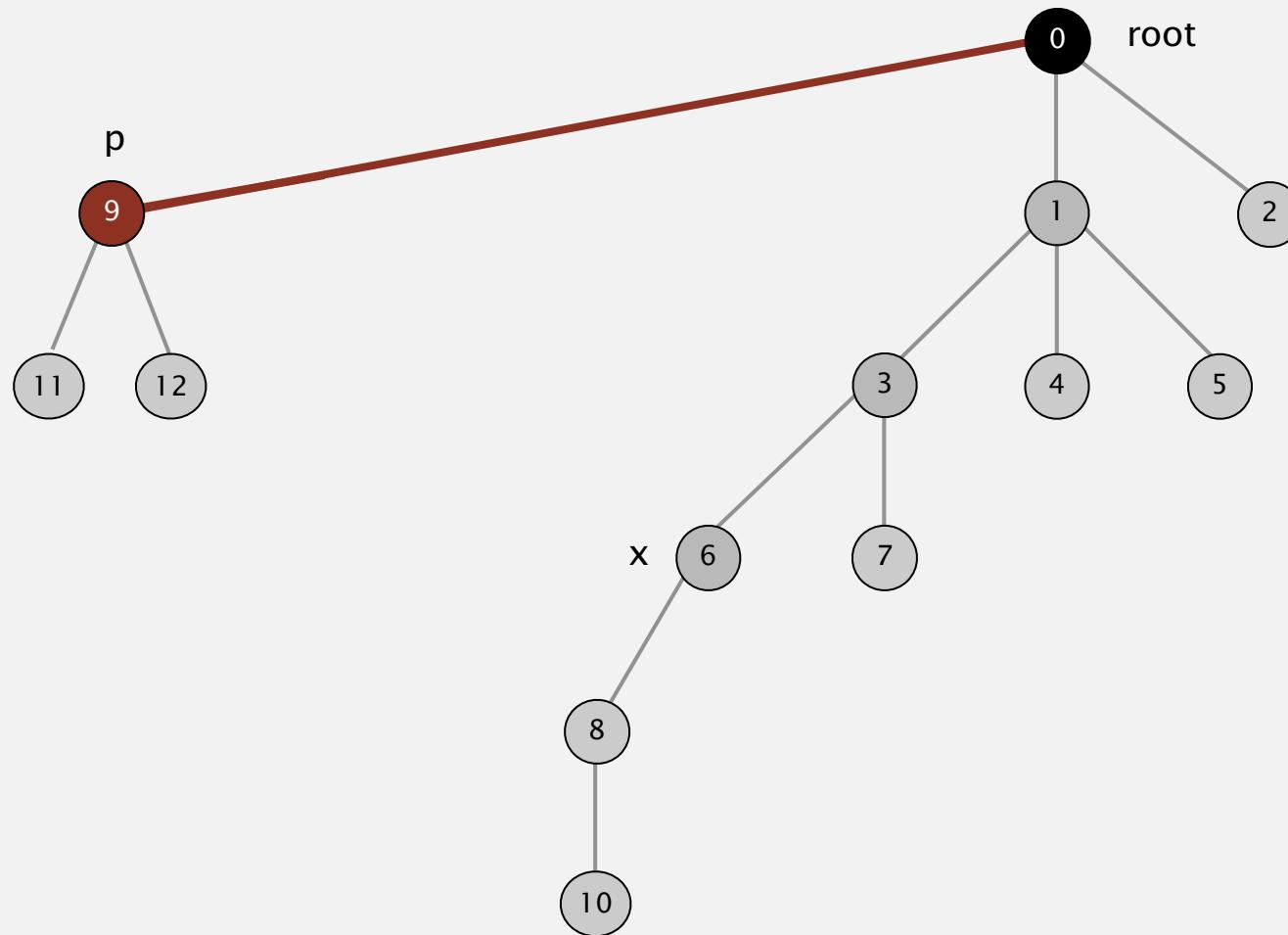
Improvement 2: path compression

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



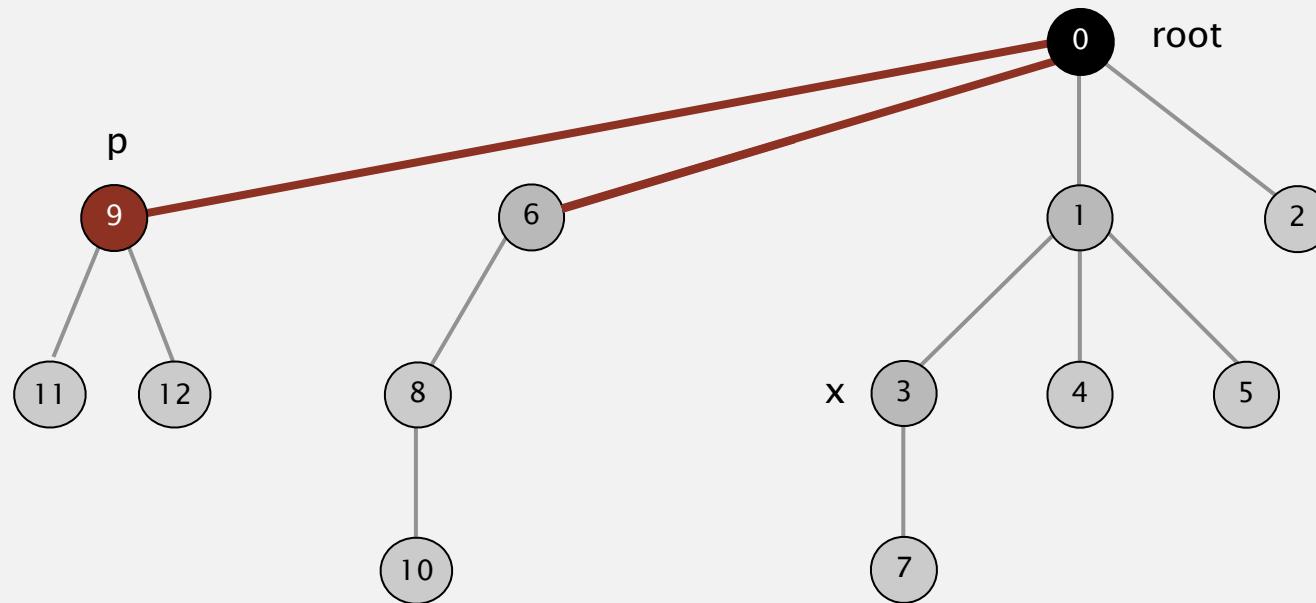
Improvement 2: path compression

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



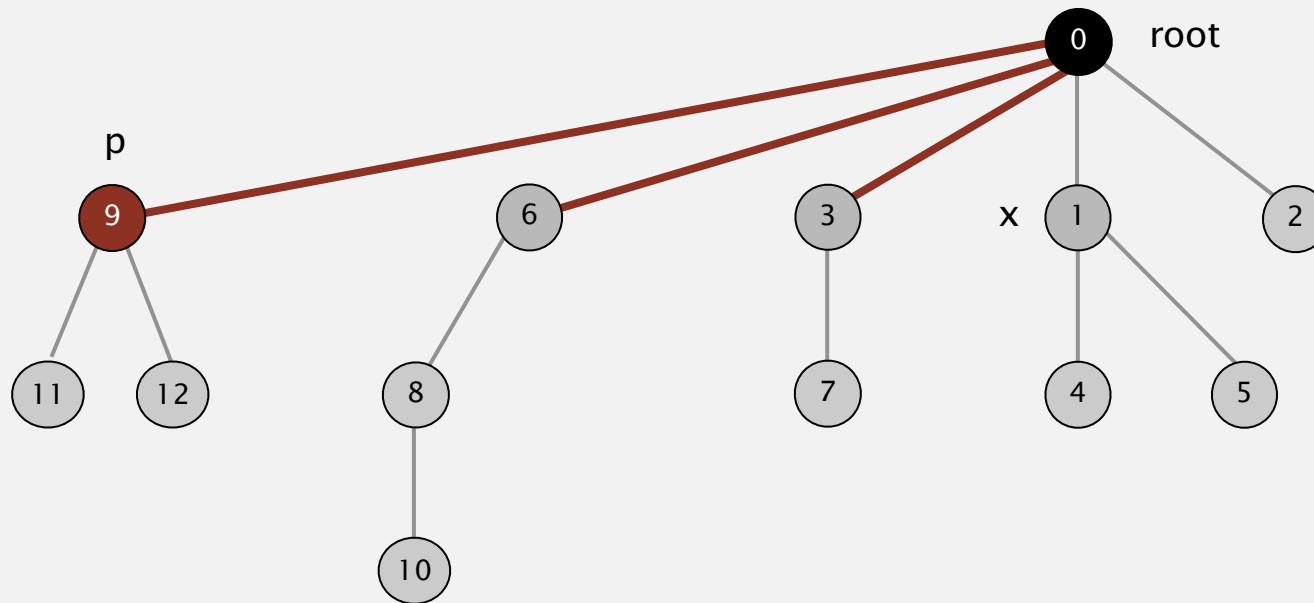
Improvement 2: path compression

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



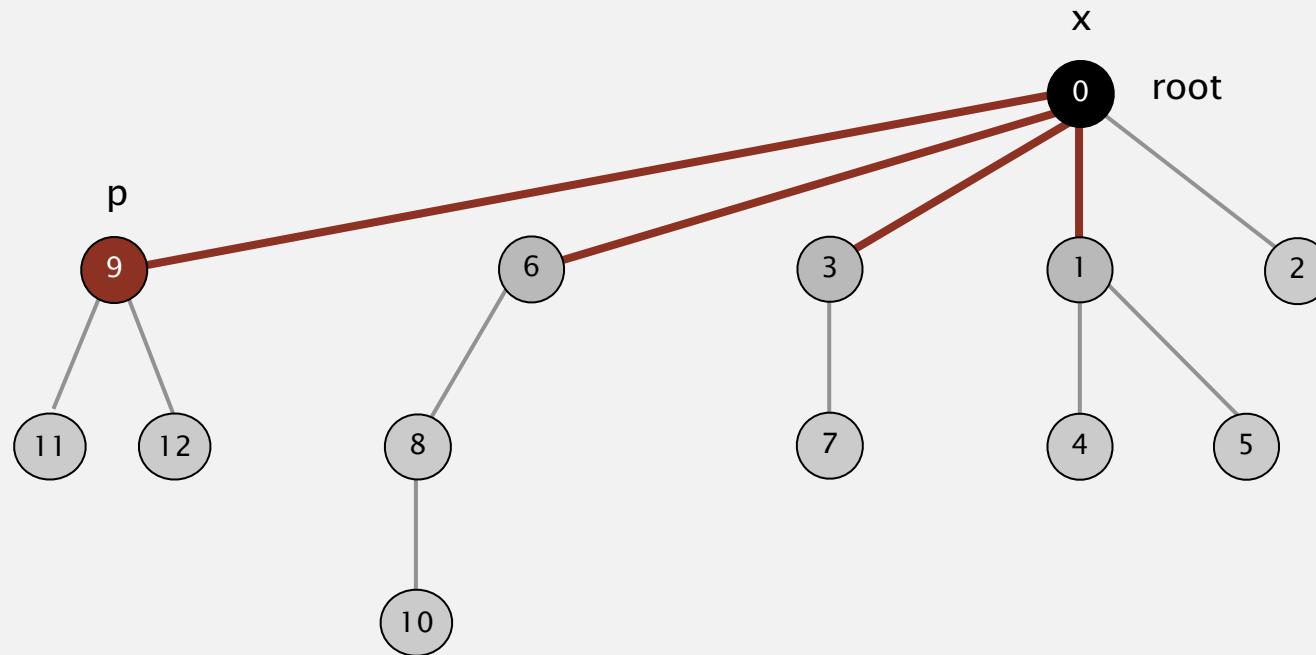
Improvement 2: path compression

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]]; ← 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: 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. [10⁹ unions and finds with 10⁹ objects]

- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

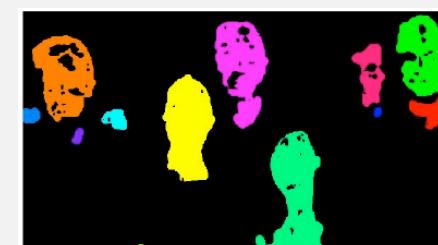
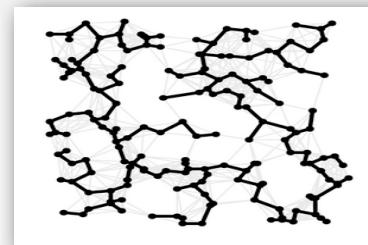
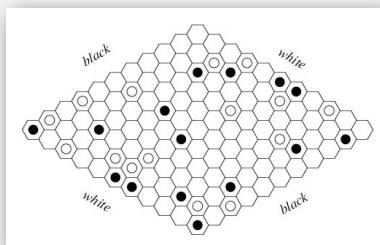
<http://algs4.cs.princeton.edu>

1.5 UNION-FIND

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

Union-find applications

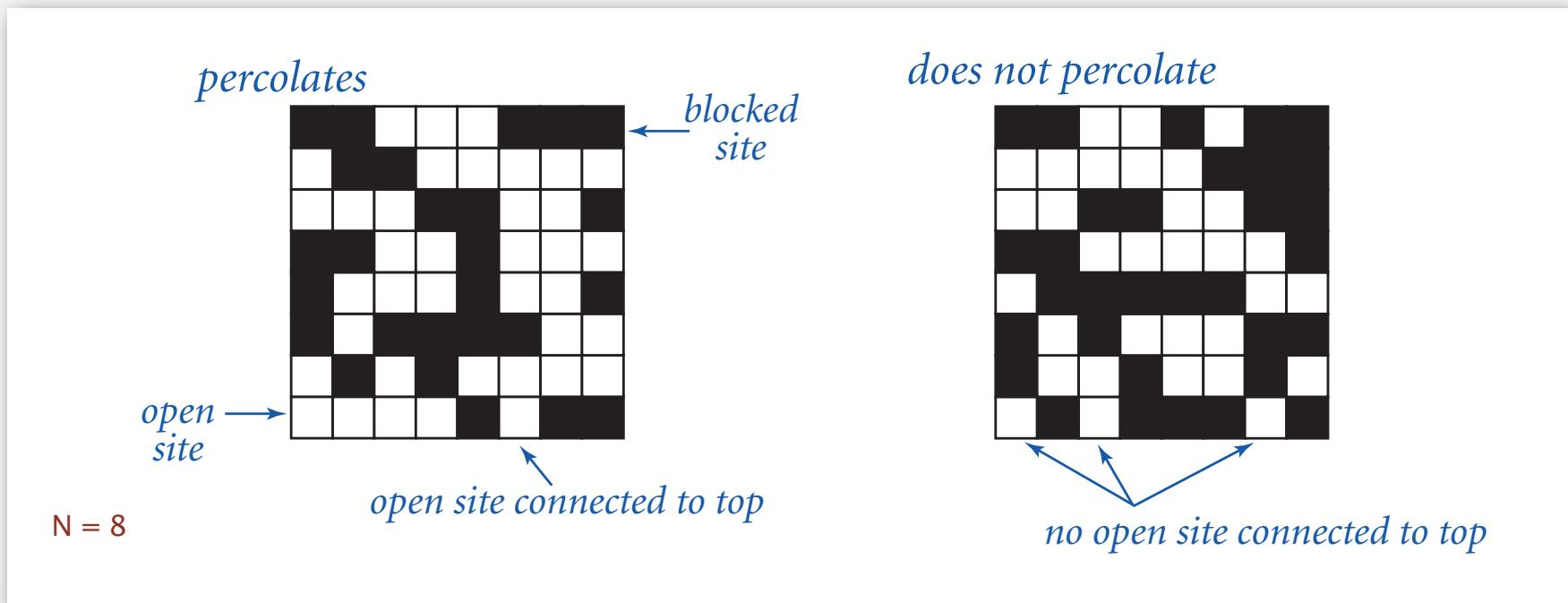
- Percolation.
- Games (Go, Hex).
- ✓ Dynamic connectivity.
 - Least common ancestor.
 - Equivalence of finite state automata.
 - Hoshen-Kopelman algorithm in physics.
 - Hinley-Milner polymorphic type inference.
 - Kruskal's minimum spanning tree algorithm.
 - Compiling equivalence statements in Fortran.
 - 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.



Percolation

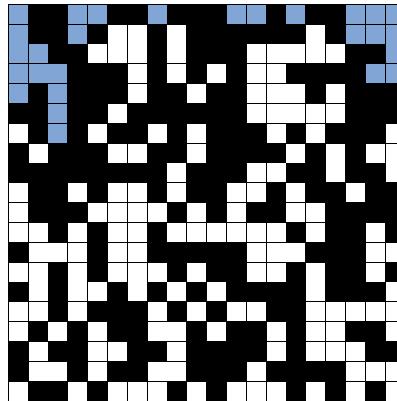
A model for many physical systems:

- 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.

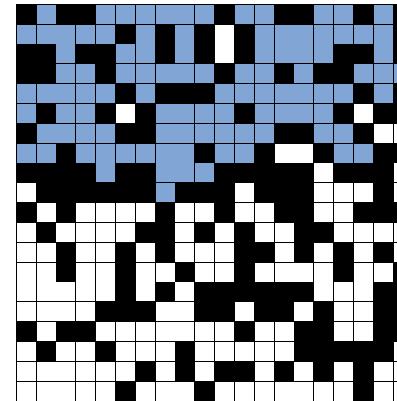
model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Likelihood of percolation

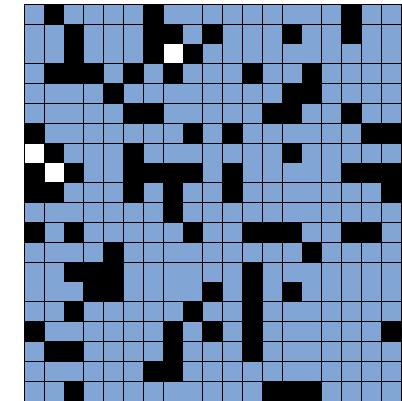
Depends on site vacancy probability p .



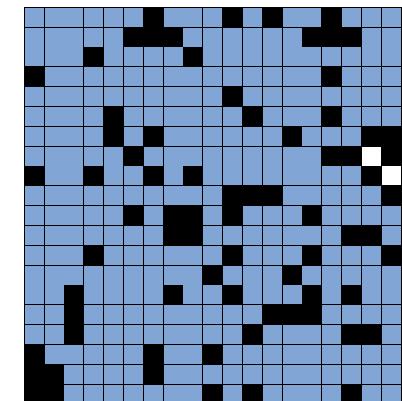
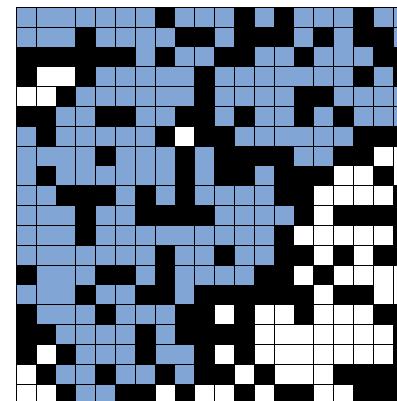
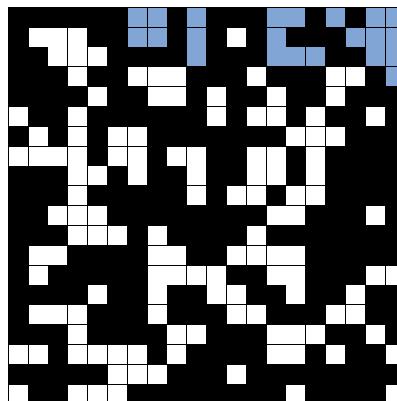
p low (0.4)
does not percolate



p medium (0.6)
percolates?



p high (0.8)
percolates

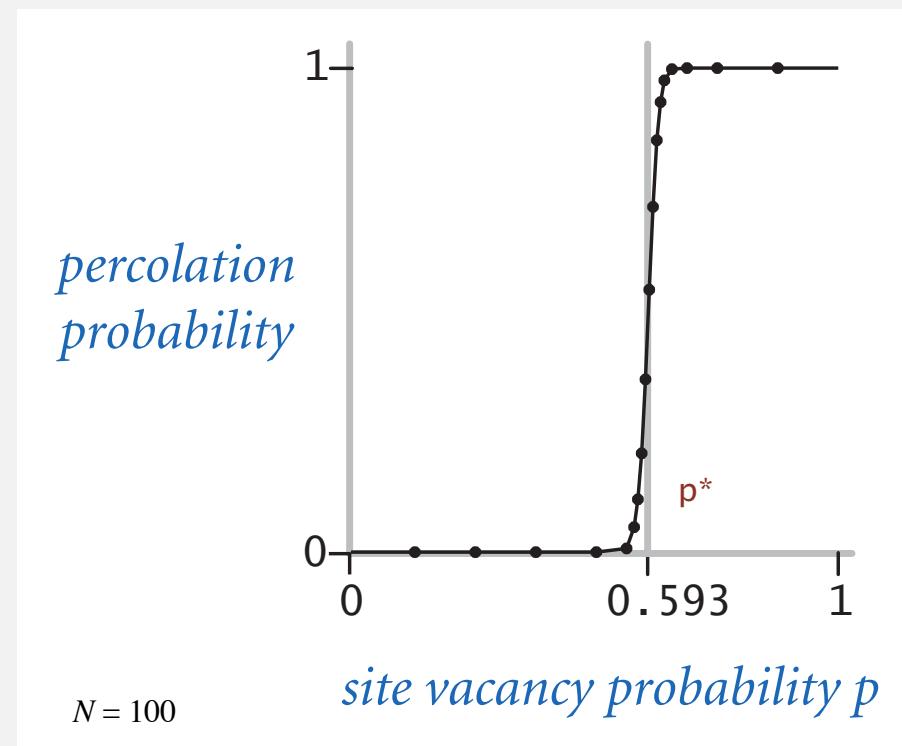


Percolation phase transition

When N is large, theory guarantees a sharp threshold p^* .

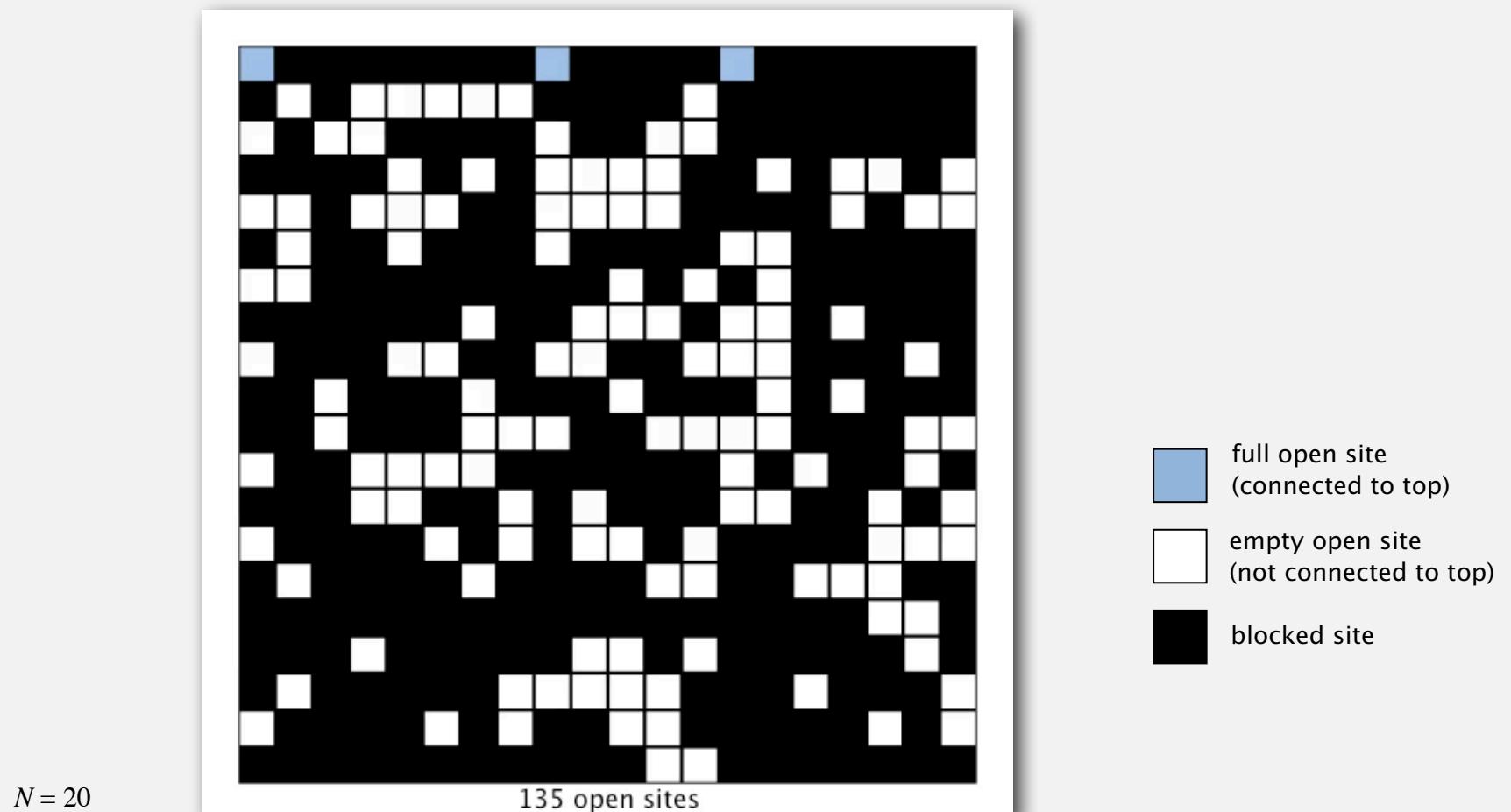
- $p > p^*$: almost certainly percolates.
- $p < p^*$: almost certainly does not percolate.

Q. What is the value of p^* ?



Monte Carlo simulation

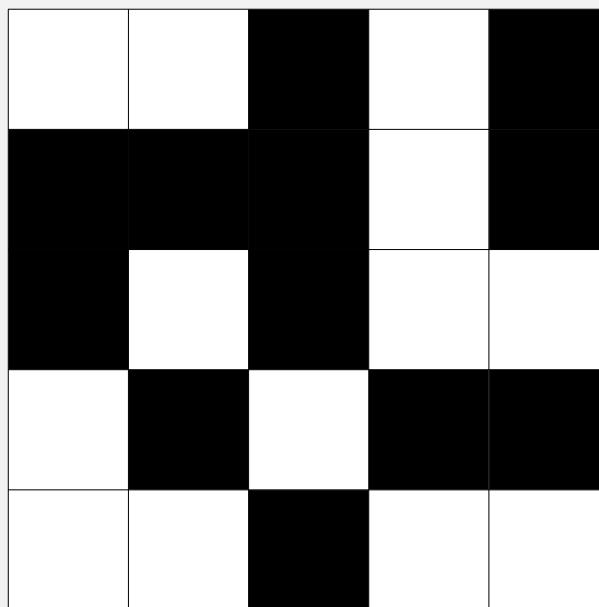
- Initialize N -by- N whole grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates p^* .



Dynamic connectivity solution to estimate percolation threshold

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

$N = 5$



open site

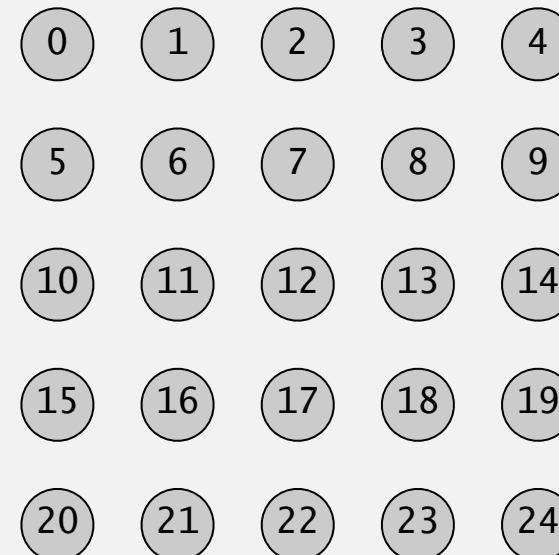
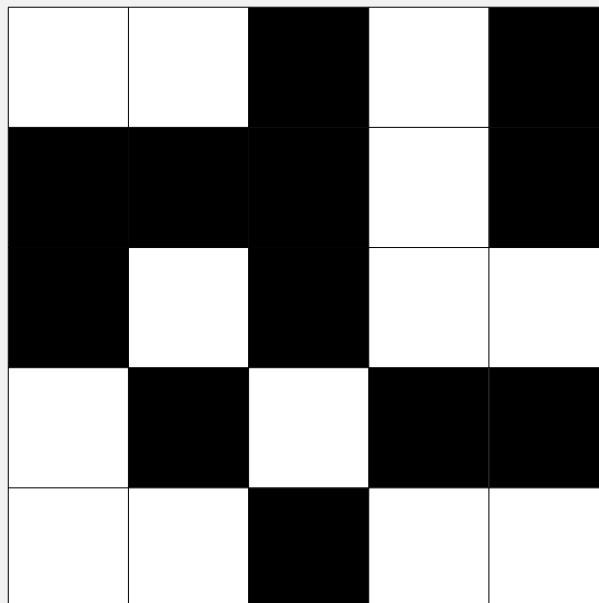
blocked site

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 $N^2 - 1$.

$N = 5$



open site

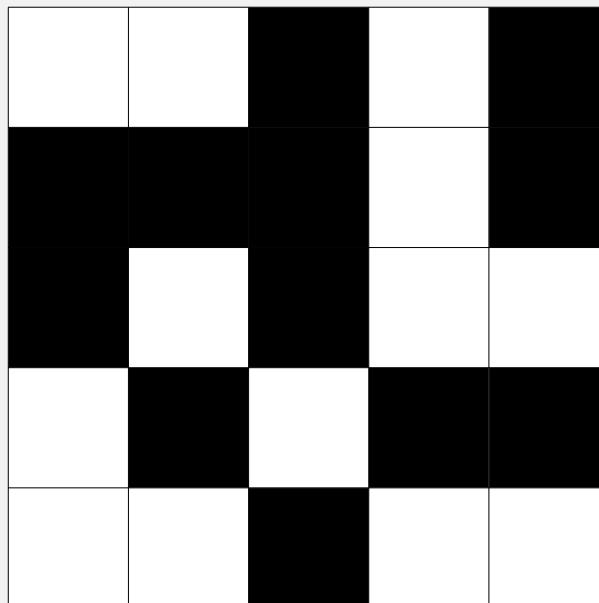
blocked site

Dynamic connectivity solution to estimate percolation threshold

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

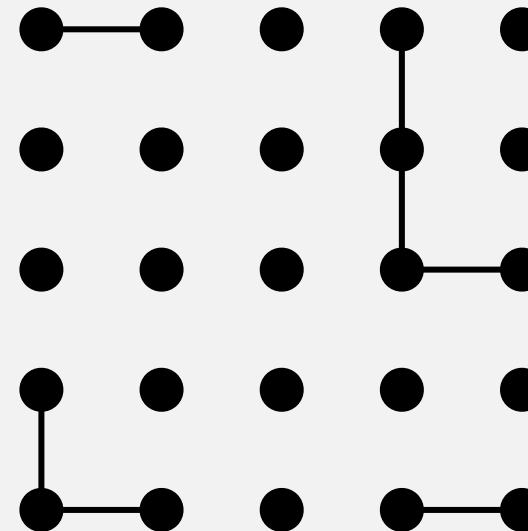
- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component if connected by open sites.

$N = 5$



open site

blocked site

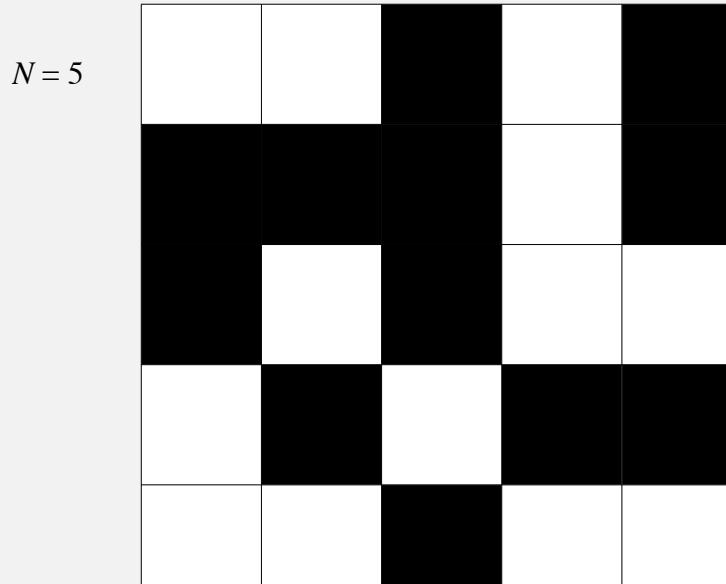


Dynamic connectivity solution to estimate percolation threshold

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

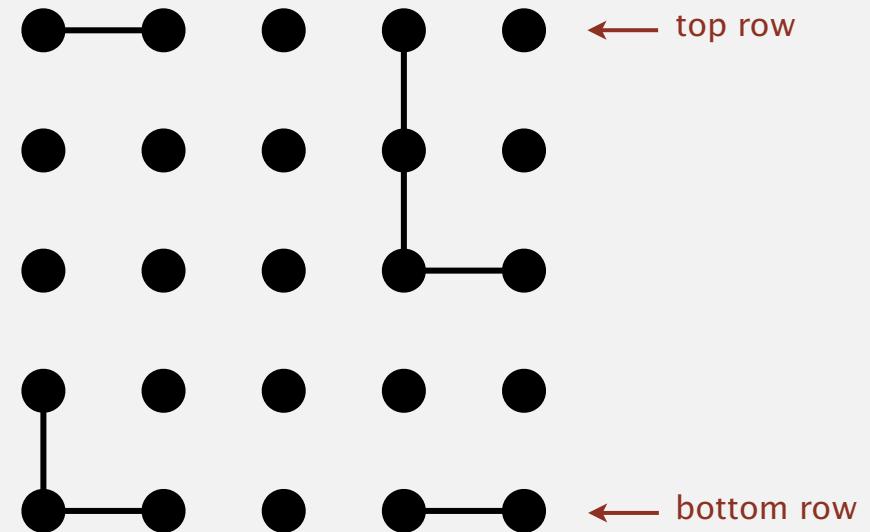
- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component if connected by open sites.
- Percolates iff any site on bottom row is connected to site on top row.

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



open site

blocked site



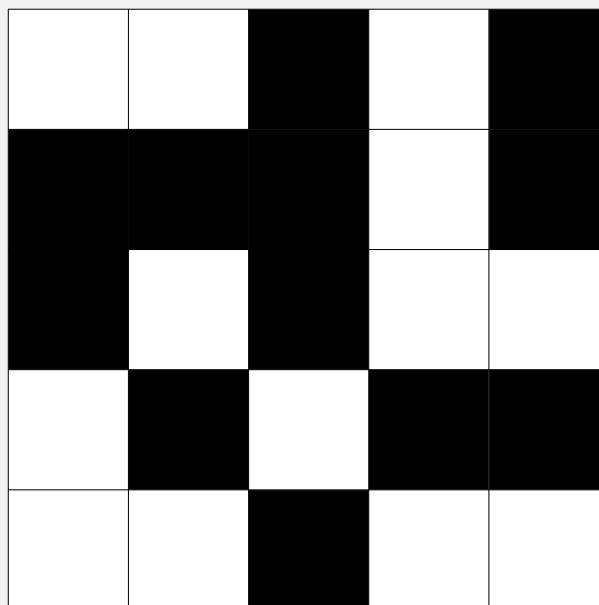
Dynamic connectivity solution to estimate percolation threshold

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

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

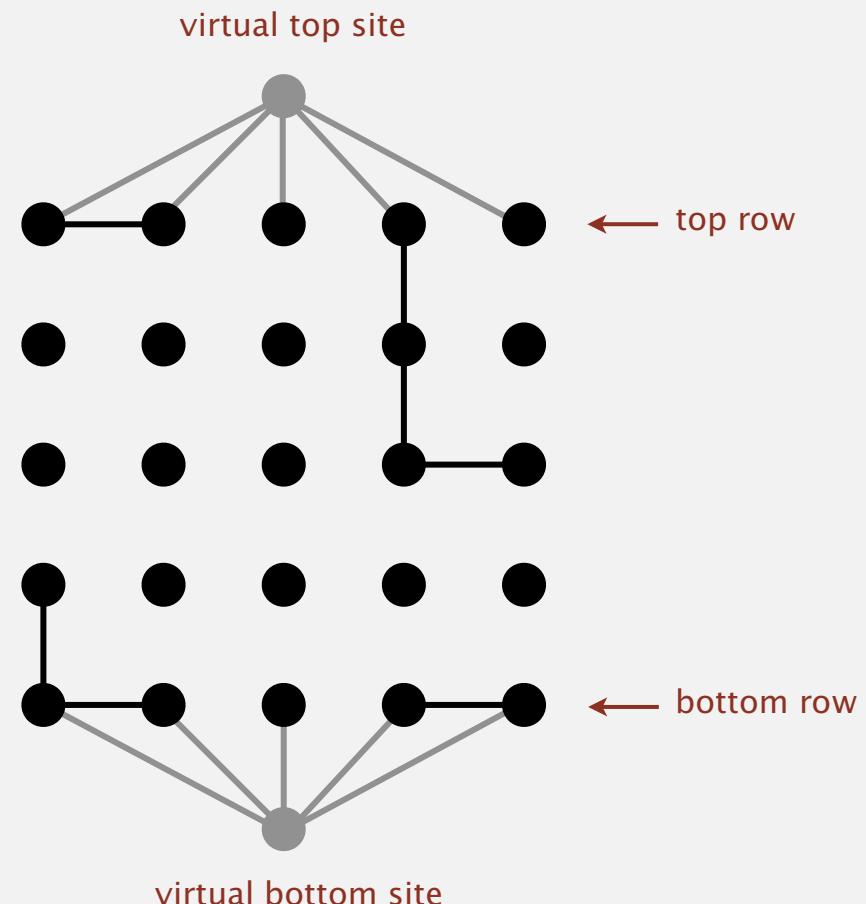
efficient algorithm: only 1 call to connected()

$N = 5$



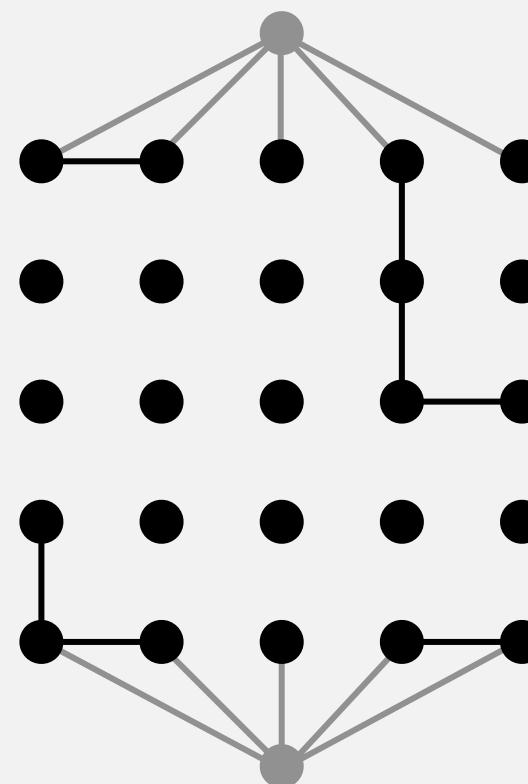
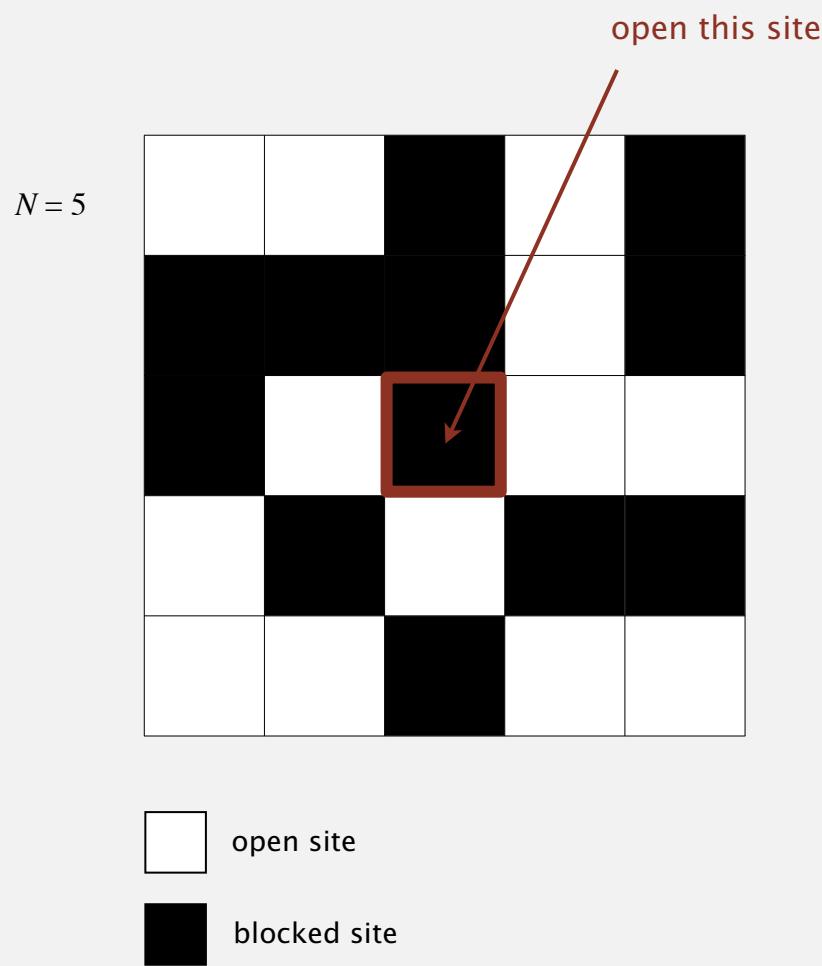
open site

blocked site



Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

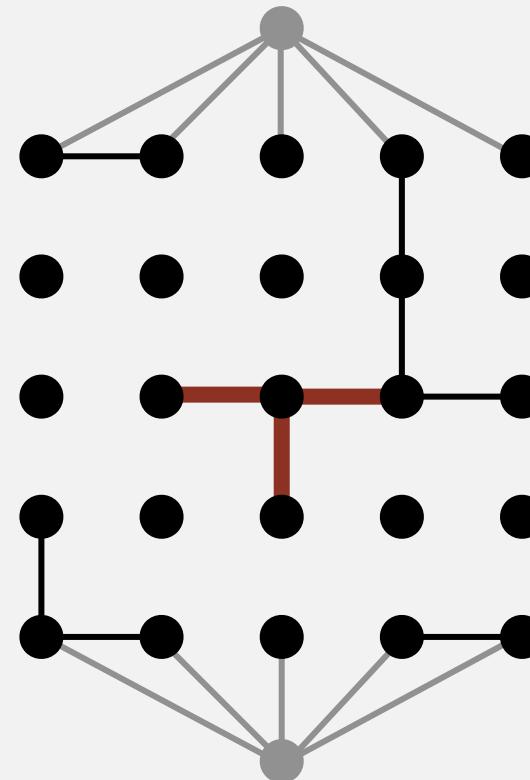
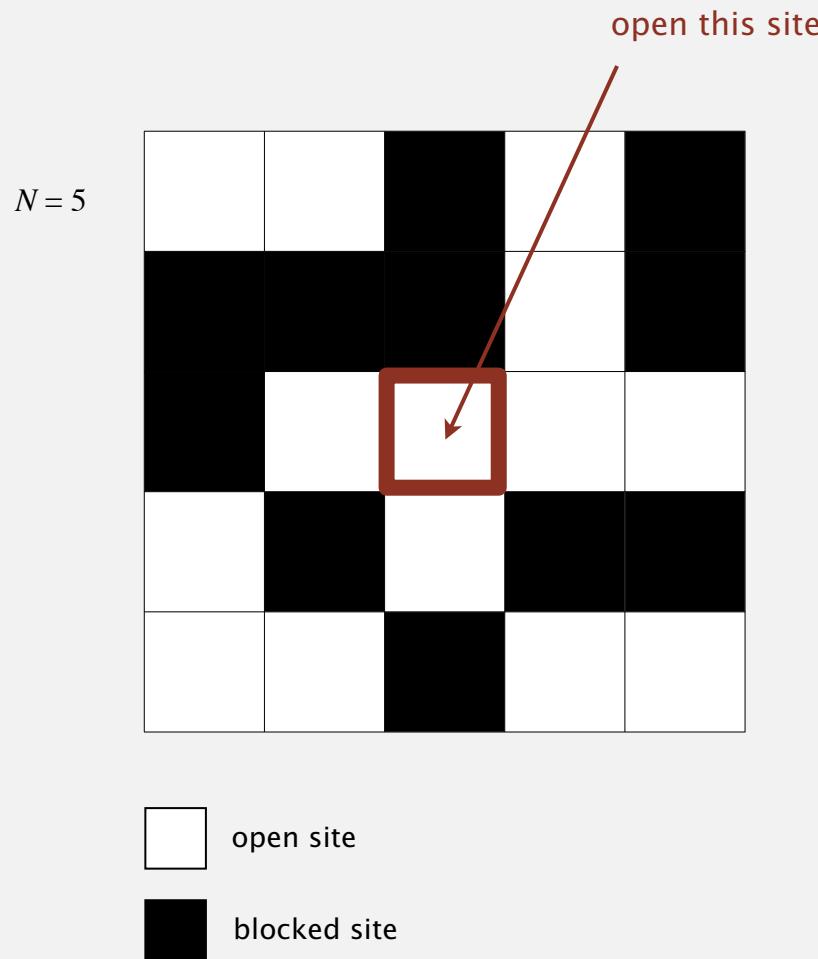


Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

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

up to 4 calls to union()

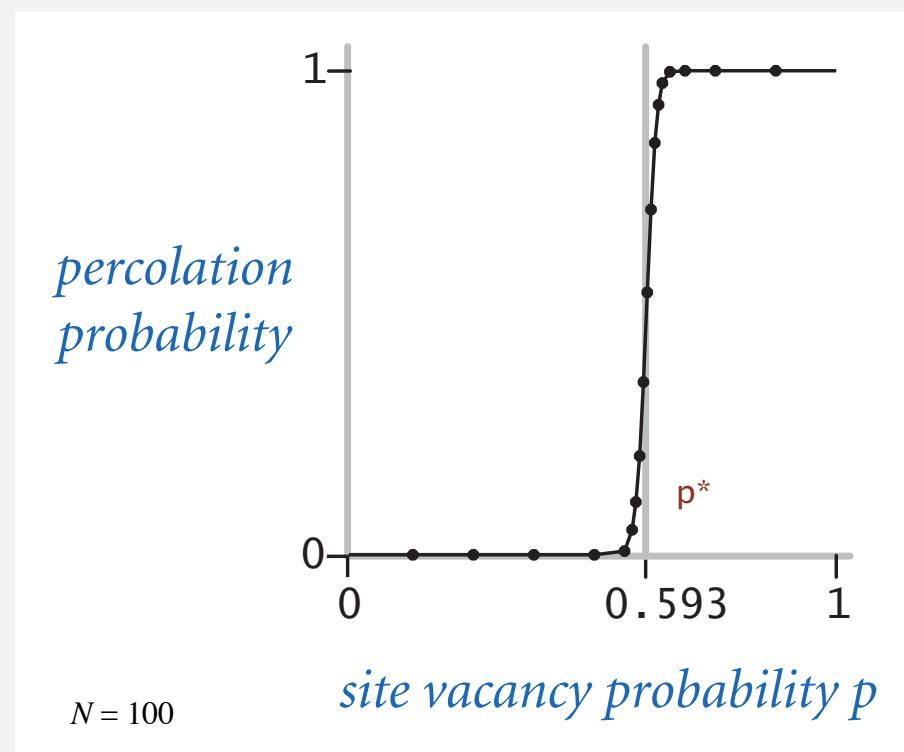


Percolation threshold

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.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<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.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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