332:573: Algorithms

INTRODUCTION

- overview
- why study algorithms?
- administrivia
- coursework
- resources
- Honesty, Plagiarism, Ethics..

573 course overview

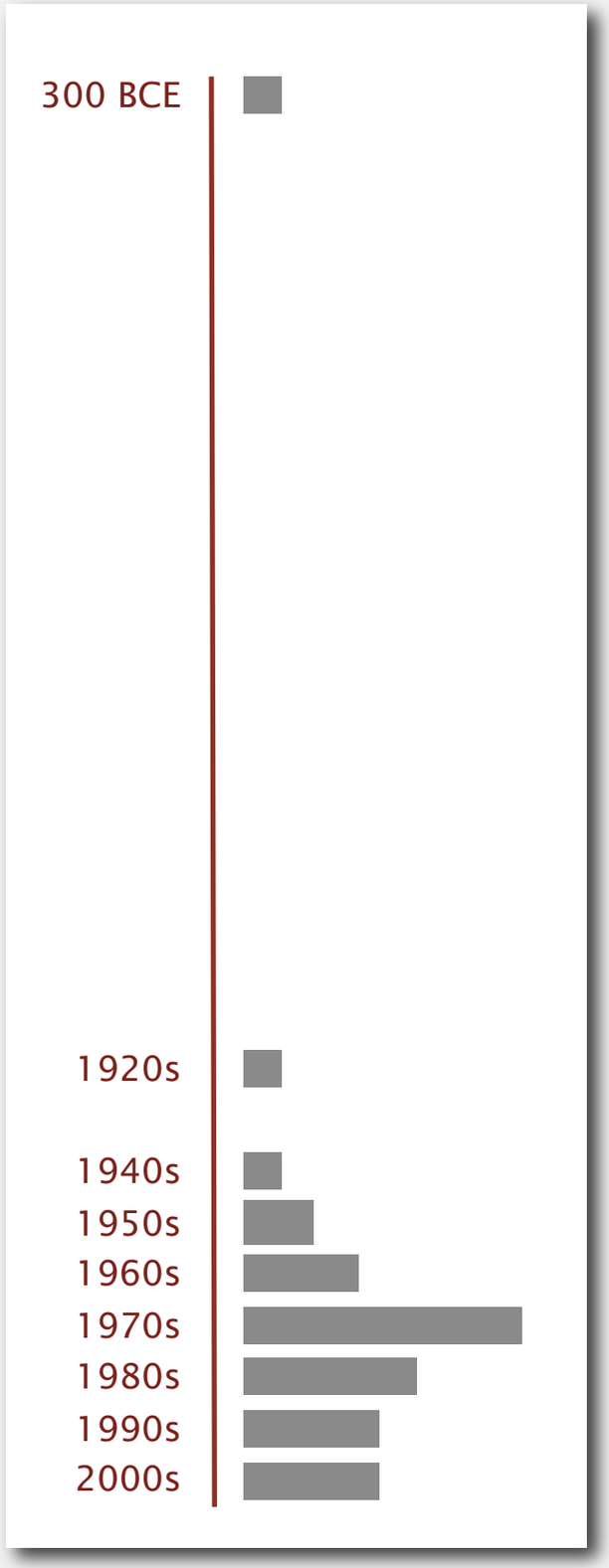
- A quick couple of definitions, to be refined several times:
  - Algorithm: methods for solving problems & programming for applications.
  - Data structure: method to store information.

topic	data structures and algorithms
fundamental	Data structures and analysis
sorting	quicksort, mergesort, heapsort, radix sorts
searching	BST, red-black BST, hash table
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	Huffman, LZW
Advanced and applied	B-tree, suffix array, maxflow, simplex

Why study algorithms?

Old roots, new opportunities.

- Study of algorithms dates at least to Euclid
- GCD (GCF) between two integers
  - 81, 27?
  - 200, 140 ?



Basis for much of science and engineering advances

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 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, ...

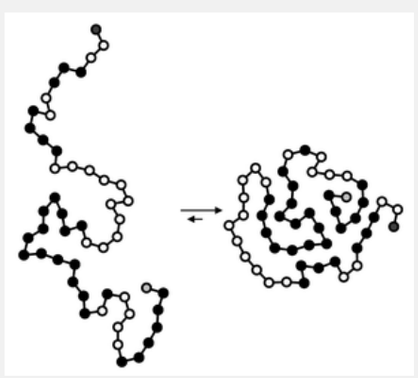
Security. Cell phones, e-commerce, voting machines, ...

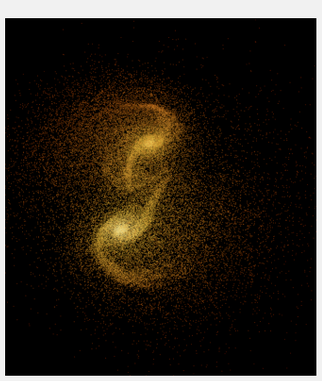
Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

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

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

...



573 course overview

What is ECE 573?

- Help understand + define the scope of this course

- What this course is not?!
  - Algos is the Greek word for pain. Algor is Latin, to be cold.

  Trivia:

  http://en.wikipedia.org/wiki/Al-Khwarizmi

  eponymous with algorithms and his book (algebra)

Administrivia

Me:

- Shantenu.jha@rutgers.edu
- Office Hours: Tuesday 1pm-2pm
  - 705 Core Building
- http://radical.rutgers.edu

Teaching Assistant: : Li Zhu lz206@scarletmail.rutgers.edu
Office Hours: Monday 1pm-2pm
Location: TBD

Grader: Aymen Al-Saadi <afa64@scarletmail.rutgers.edu>
Will not have regular office hours but can be met by appointment

Coursework and grading

Homeworks 15% [3 x 5 pts each][best 3 out of 4]

- Due before class starts (electronic submission)
- Where applicable please use PDF

Three Mid-term Exams: 45%

[Estimated dates: 2nd last week of Feb, Last week of March, 3rd Last class]

Quizzes: 15% (3 x 5pts)

Term Paper: 10%

Deadline: Due on April 01.

Project: 15%

- We will discuss possible ideas + scope soon
- Proposals (title and abstract) are due before spring break
- Written Report (due on last class)
- Project Presentations (last two classes)

Analyzing and improving Algorithm [useful for class Project]

Steps to developing a usable algorithm.

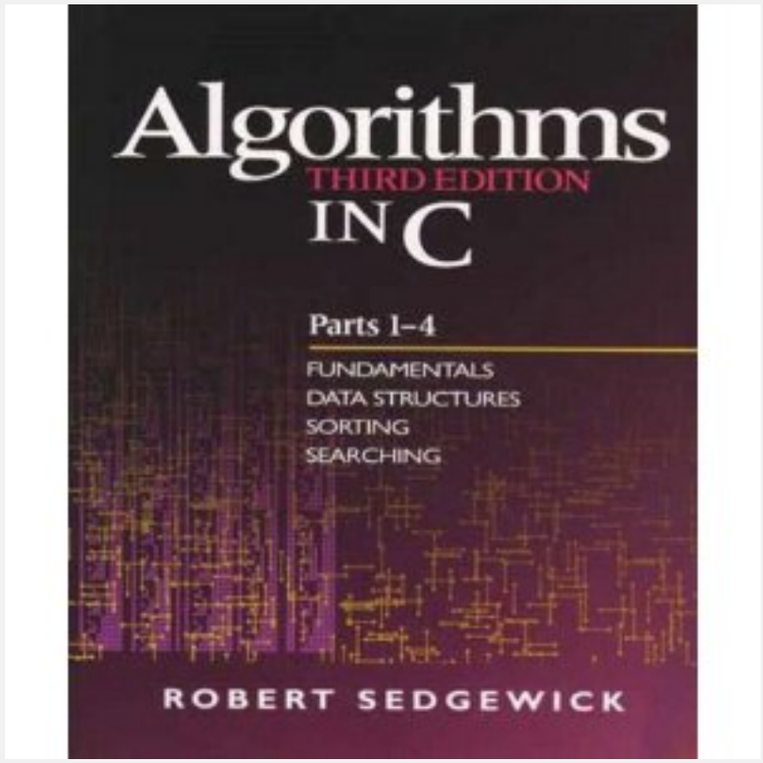
- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

Resources (textbook)

Textbook: Algorithms 3rd edition by R. Sedgewick, Addison-Wesley Professional.



Resources

I will also be using the following book: Algorithms 4th edition by R. Sedgewick and K. Wayne, Addison-Wesley Professional, 2011, ISBN 0-321-57351-X.

Pedagogically far superior than the Algorithms in C++/Java/C series

emphasis on algorithms as an empirical tool and not just theoretical !!

applications and examples

Very similar in coverage but at a different level

ACKNOWLEDGE MANY SLIDES FROM THE BOOKS WEBSITE

http://algs4.cs.princeton.edu/home/

Strongly advised to visit book's website and check out slides, examples, demos etc. However this is not the text for the course

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

Before we proceed any further…

Integrity, Honesty and Ethics

Academic Integrity Policy

http://satest2.rutgers.edu/files/documents/AI_Policy_9_01_2011.pdf

I WILL STRICTLY ENFORCE THE HIGHEST STANDARDS

THERE WILL BE NO SECOND CHANCES

Principles and Practise of the Analysis of Algorithms

- **Perspective**
- **observations**
- **mathematical models**
- **order-of-growth classifications**
- **dependencies on inputs**
- **memory**

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

Primary practical reason: avoid performance bugs.

Top 10 Algorithms of the Century:

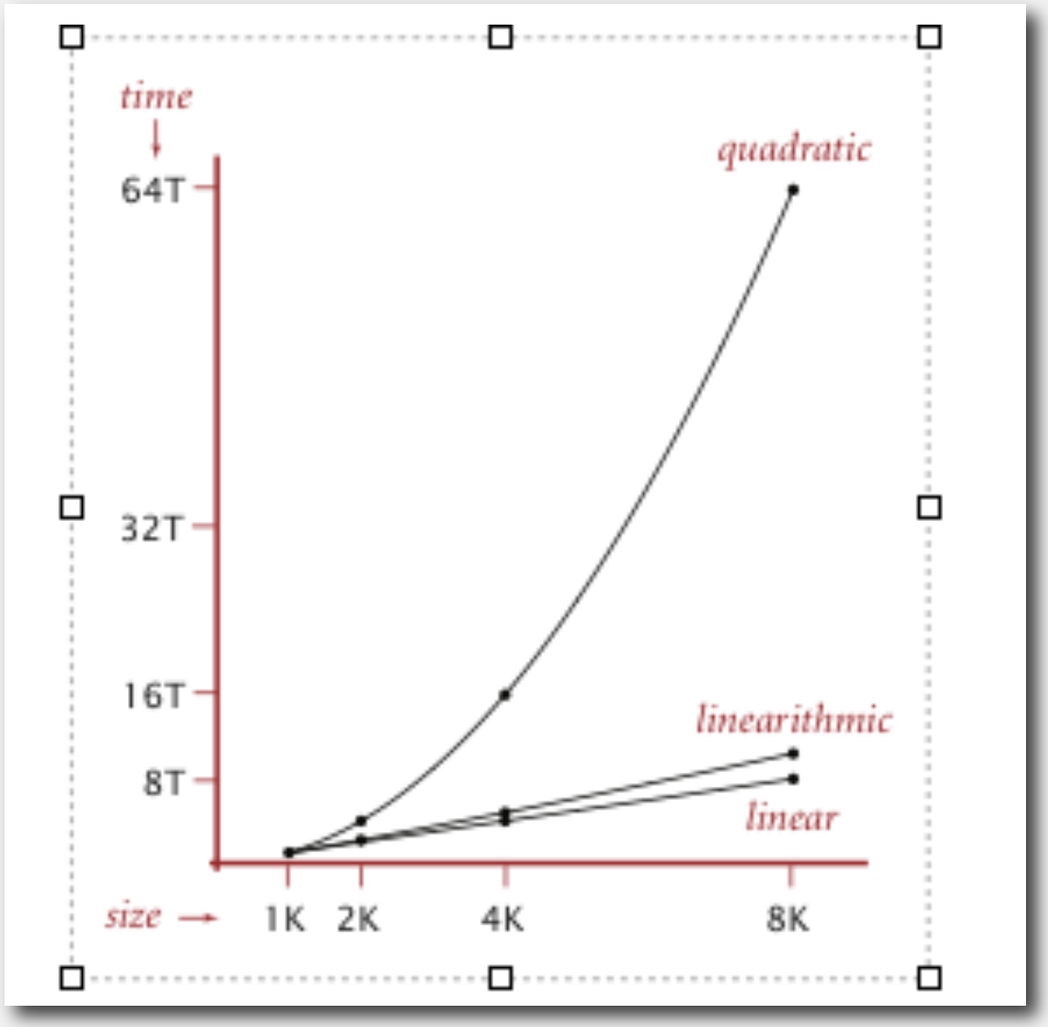
http://www.siam.org/pdf/news/637.pdf

FMM
DFT
Monte Carlo Methods
Quicksort..

Some algorithmic successes

N-body simulation.

- Simulate gravitational interactions among N bodies.
- Brute force: N^2 steps.
- N-Body Methods:
  - Barnes-Hut algorithm: $N \log N$ steps, enables new research.
  - Fast Multiple Methods (FMM): O(N)

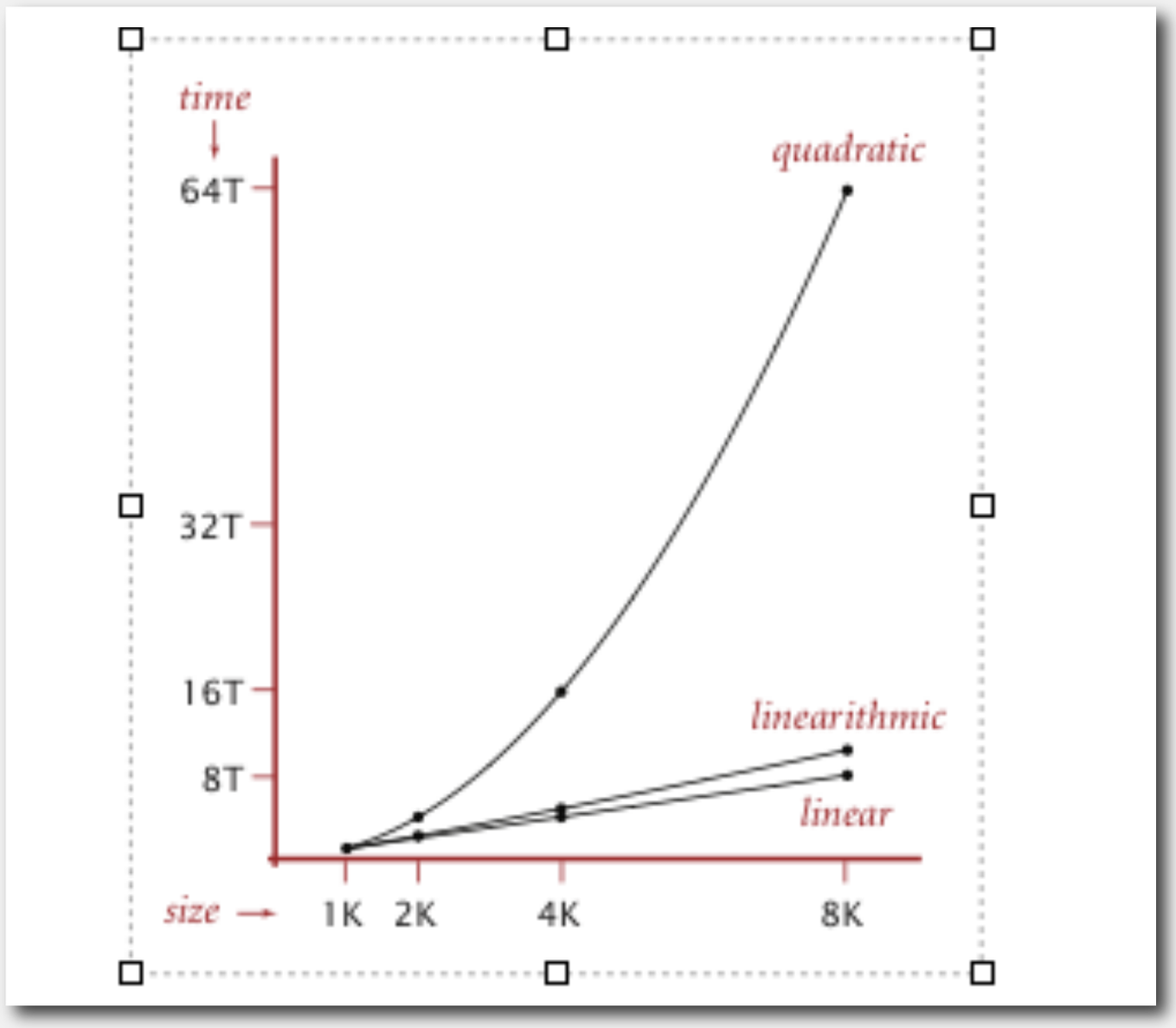


Some algorithmic successes

Discrete Fourier transform.

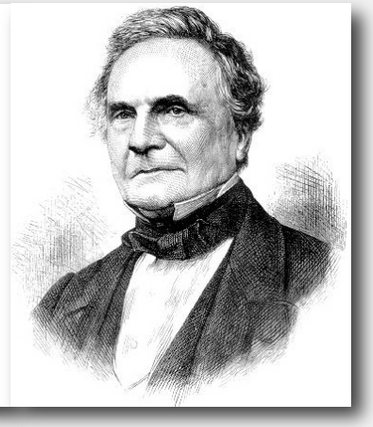
- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics, ….
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, enables new technology.

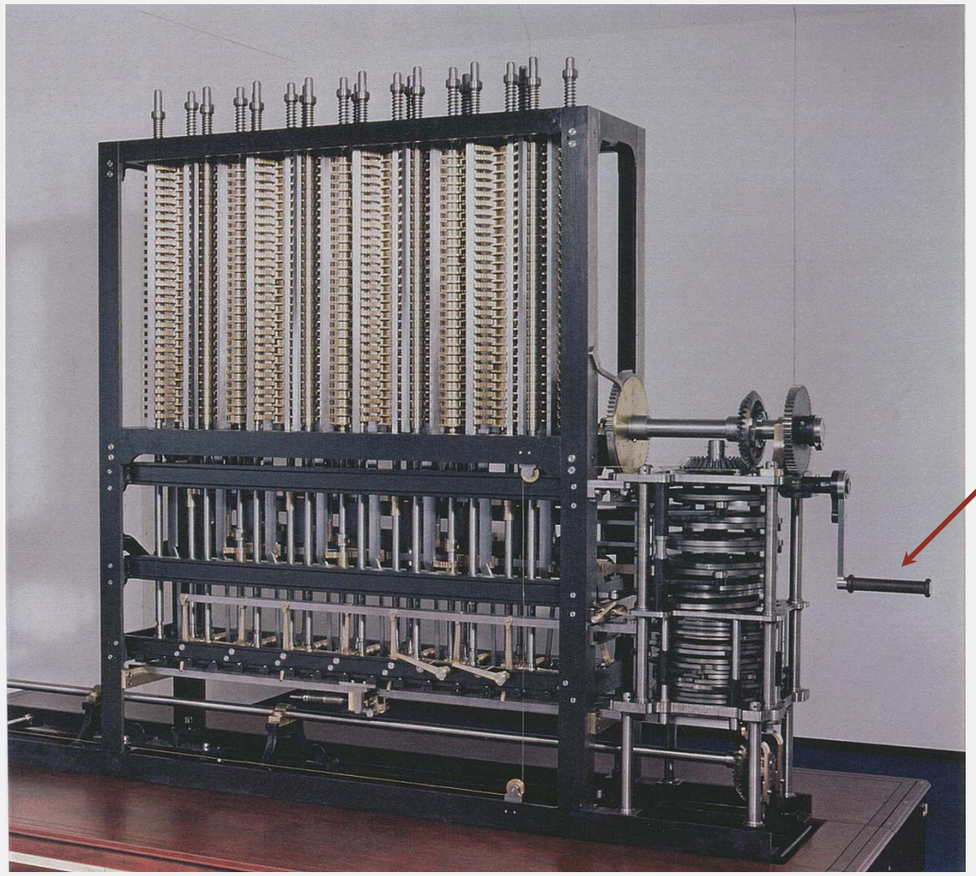
Friedrich Gauss
1805



Slightly more rigorous approach than…

"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

Scientific method applied to analysis of algorithms

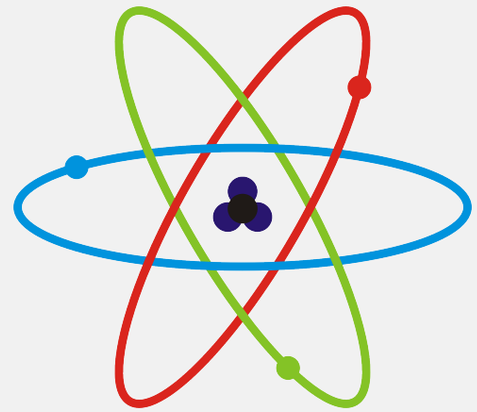
A framework for predicting performance and comparing algorithms.

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.



Algorithmics (= the science of algorithms) more than just algorithms!

Measuring the Efficiency of Algorithms

Analysis of algorithms

- Provides tools for contrasting the efficiency of different methods of solution
  - Time efficiency, space efficiency
  - Focus is on time efficiency

A comparison of algorithms

- Should focus on significant differences in efficiency
- Should not consider reductions in computing costs due to clever coding tricks

Measuring the Efficiency of Algorithms

Three difficulties with comparing programs instead of algorithms

- How are the algorithms coded?
  - Distinguish implementation from the Algorithm!
- What computer should you use?
  - HW specific features!
- What data should the programs use?
  - E.g. sequential search and smallest data at top!

Algorithm analysis should be independent of

- Specific implementations
- Computers
- Data

The Execution Time of Algorithms

Counting an algorithm's operations is a way to assess its time efficiency

- An algorithm's execution time is related to the number of operations it requires
- Example: Traversal of a linked list of n nodes
  - $n + 1$ assignments, $n + 1$ comparisons, n writes
- Example: The Towers of Hanoi with n disks
  - $2^n - 1$ moves

Algorithm Growth Rates

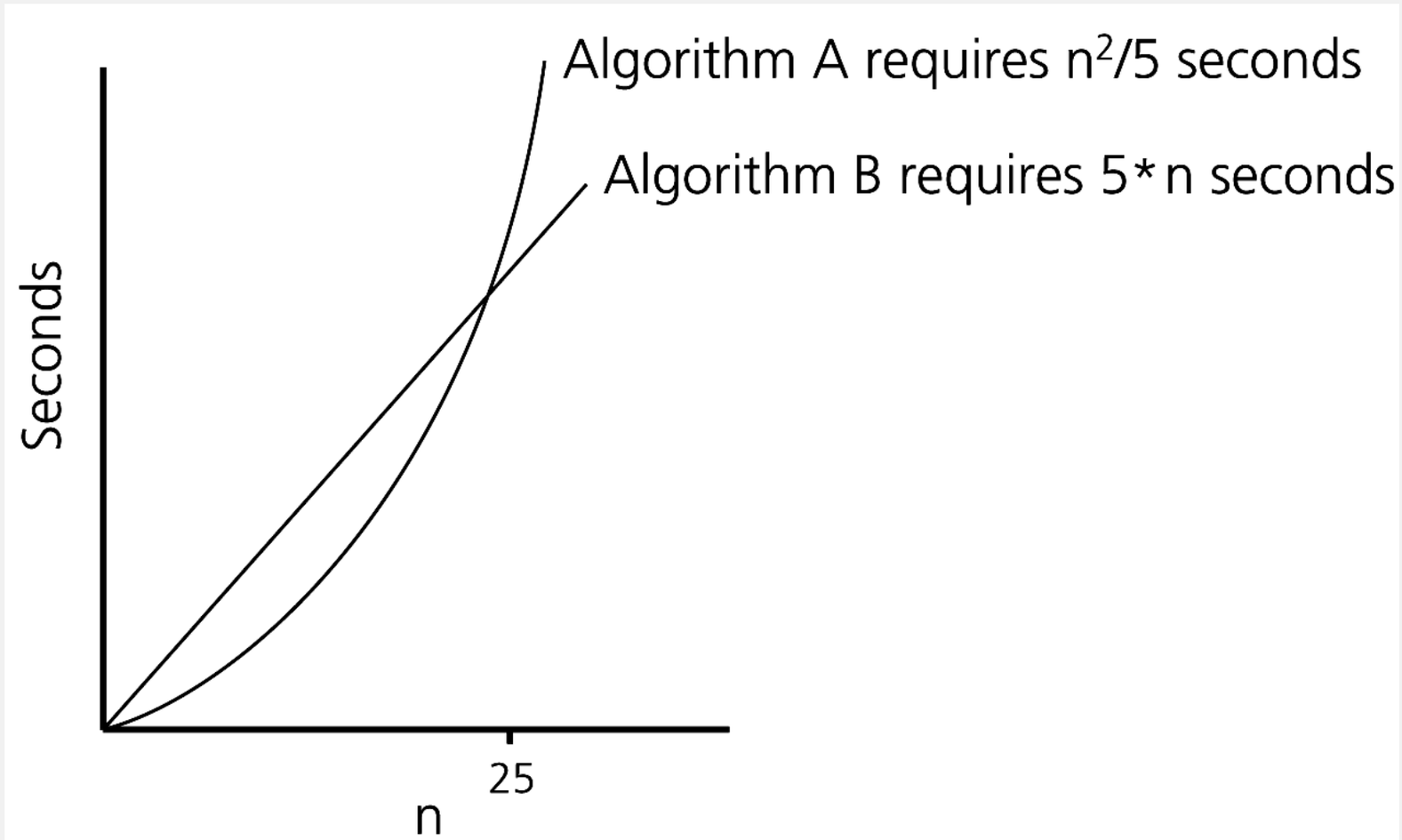
An algorithm's time requirements can be measured as a function of the problem size

- Number of nodes in a linked list
- Size of an array
- Number of items in a stack
- Number of disks in the Towers of Hanoi problem

Algorithm efficiency is typically a concern for large problems only

Algorithm Growth Rates

Time requirements as a function of the problem size *n*



- Algorithm A requires time proportional to n^2
- Algorithm B requires time proportional to n

Algorithm Growth Rates

- An algorithm's growth rate
  - Enables the comparison of one algorithm with another
  - Algorithm A requires time proportional to n^2
  - Algorithm B requires time proportional to n
  - Algorithm B is faster than Algorithm A

- n^2 and n are growth-rate functions
- Algorithm A is $O(n^2)$ - order n^2
- Algorithm B is $O(n)$ - order n

- Big O notation

Order-of-Magnitude Analysis and Big O Notation

Definition of the order of an algorithm

Algorithm *A* is order $f(n)$ – denoted $O(f(n))$ – if constants k and n_0 exist such that *A* requires no more than $k * f(n)$ time units to solve a problem of size $n \geq n_0$

Growth-rate function $f(n)$

- A mathematical function used to specify an algorithm's order in terms of the size of the problem

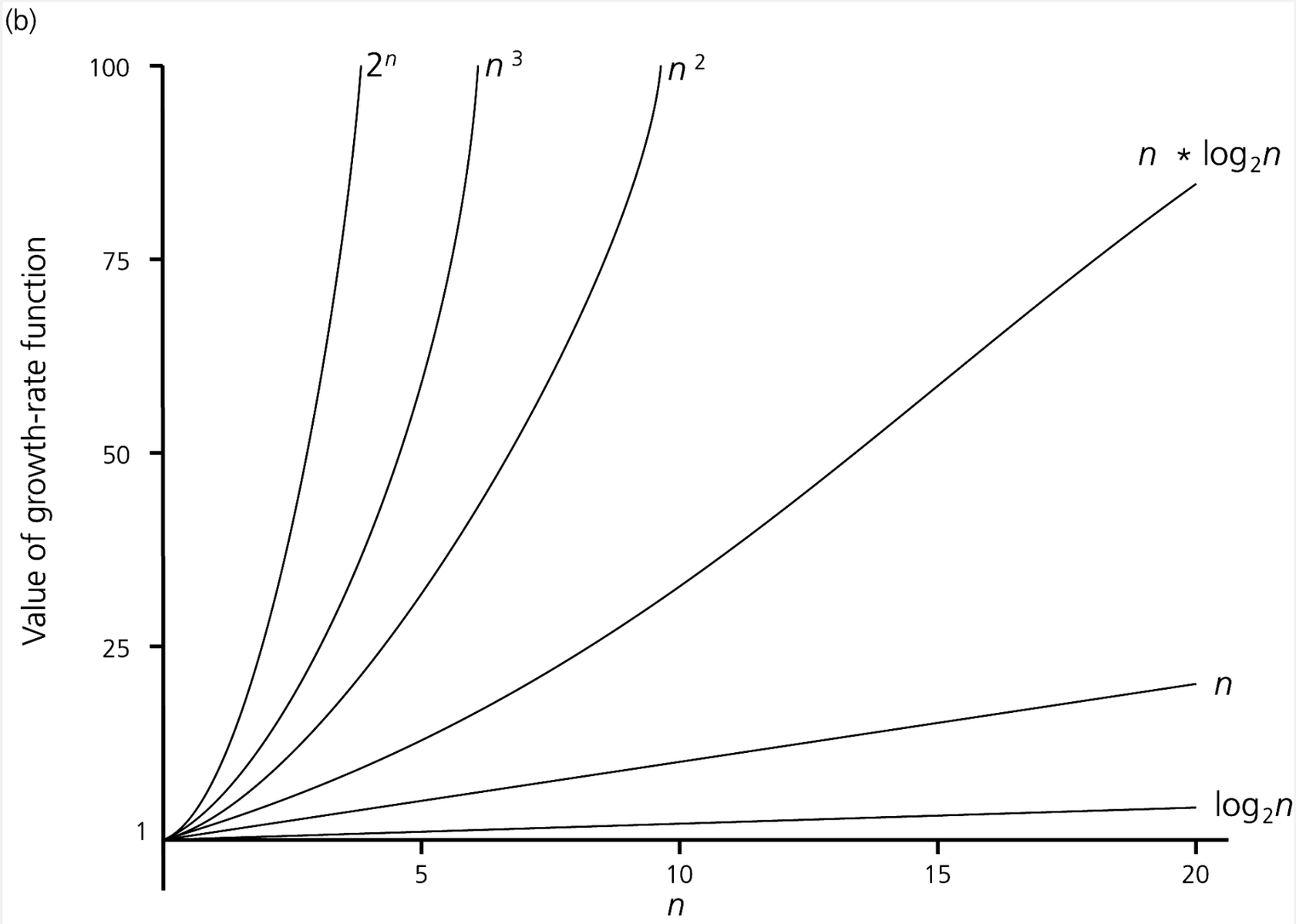
Order-of-Magnitude Analysis and Big O Notation

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A comparison of growth-rate functions in tabular form

Order-of-Magnitude Analysis and Big O Notation



A comparison of growth-rate functions in graphical form

Order-of-Magnitude Analysis and Big O Notation

Order of growth of some common functions

- $O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

Properties of growth-rate functions

- $O(n^3 + 3n)$ is $O(n^3)$: ignore low-order terms
- $O(5\ f(n)) = O(f(n))$: ignore multiplicative constant in the high-order term
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Order-of-Magnitude Analysis and Big O Notation

Worst-case analysis

- A determination of the maximum amount of time that an algorithm requires to solve problems of size *n*
  - *Quicksort on ascending sorted data*

Average-case analysis

- A determination of the average amount of time that an algorithm requires to solve problems of size *n*

Best-case analysis

- A determination of the minimum amount of time that an algorithm requires to solve problems of size *n*

Keeping Your Perspective

Only significant differences in efficiency are interesting

Frequency of operations

- When choosing an ADT's implementation, consider how frequently particular ADT operations occur in a given application
  - Array-based retreive is faster than pointer-based
  - For insertions pointer-based
- However, some seldom-used but critical operations must be efficient
  - Airline collision detection operation/algorithm

Keeping Perspective

If the problem size is always small, you can probably ignore an algorithm's efficiency

- Order-of-magnitude analysis focuses on large problems
- Figure where n > n^2

Weigh the trade-offs between an algorithm's time requirements and its memory requirements

Compare algorithms for both style and efficiency

Principles and Practise of the Analysis of Algorithms

- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- 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++)
               if (a[i] + a[j] + a[k] == 0)
                  count++;
      return count;
   }

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

check each triple

for simplicity, ignore integer overflow

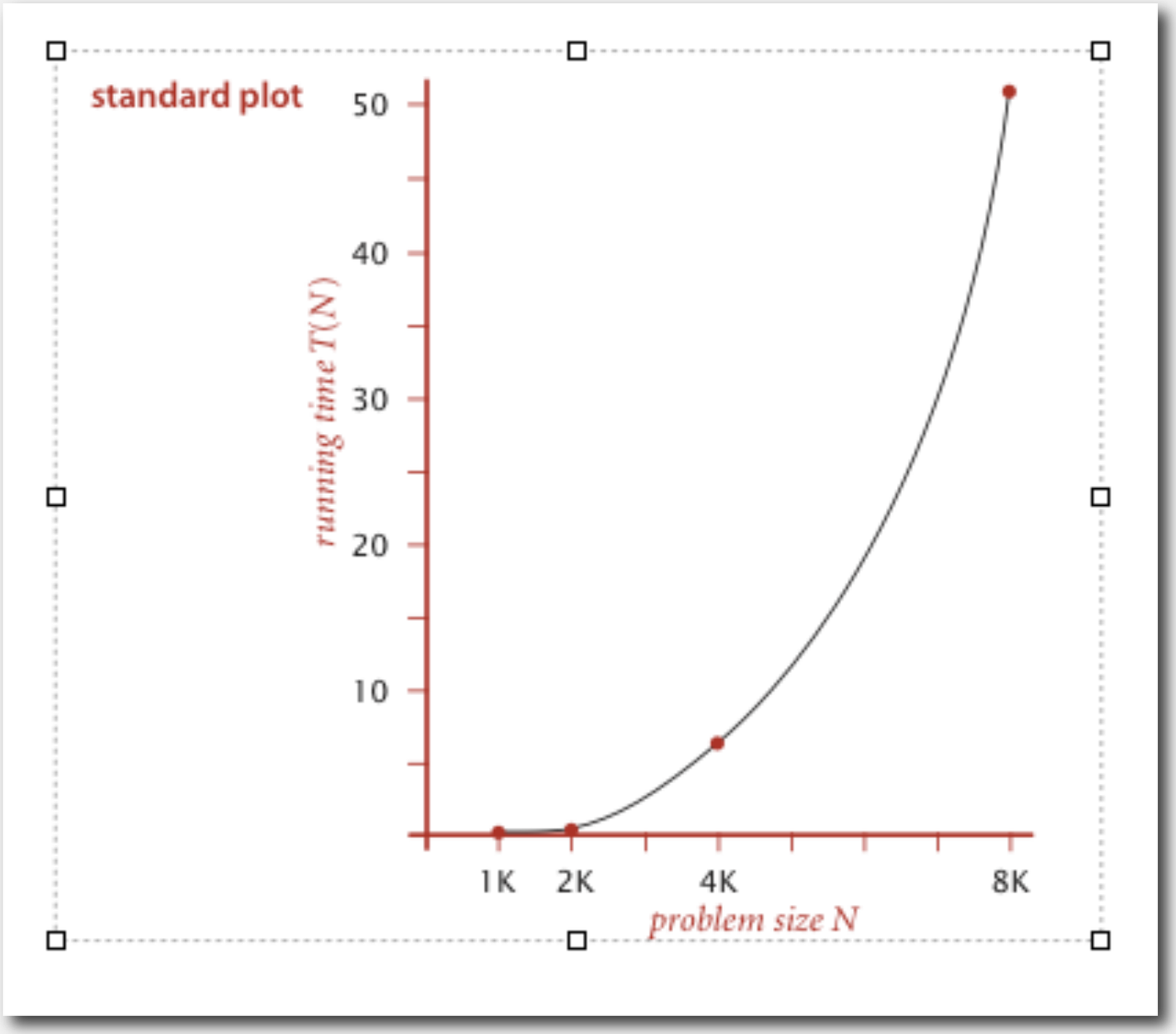
Empirical analysis

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	?

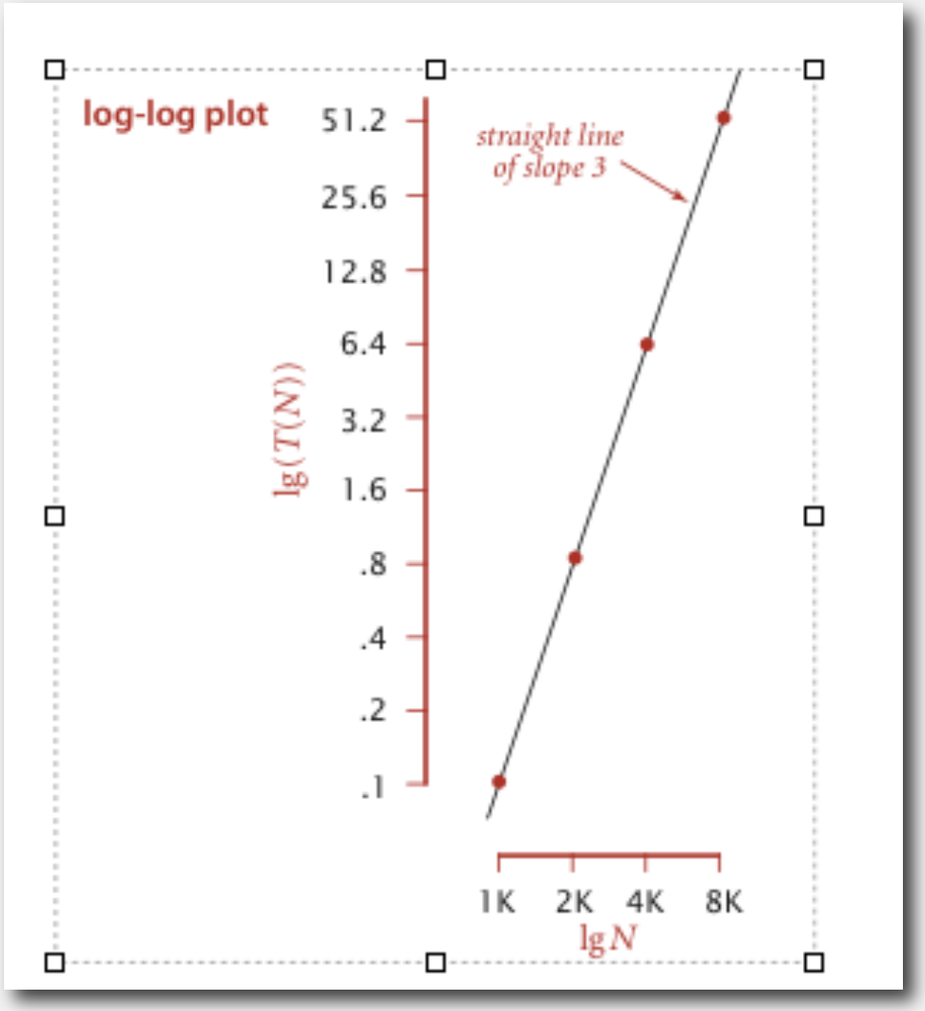
Data analysis

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.



$\lg(T(N)) = b \lg N + c$

$b = 2.999$

$c = -33.2103$

$T(N) = a N^b$, *where* $a = 2^c$

Regression. Fit straight line through data points: $a N^b$. (power law; slope)

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

Prediction and validation

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

"order of growth" of running time is about N^3

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.

ratio = T(2000) / T(1000) ≈ 8

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

lg ratio ≈ 3

seems to converge to a constant b ≈ 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 hypothesis.

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

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

$$\Rightarrow \quad a = 0.998 \cdot 10^{-10}$$

Hypothesis. Running time is about $0.998 \cdot 10^{-10} \cdot N^3$ 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 applications, ...

} helps determines constant a in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

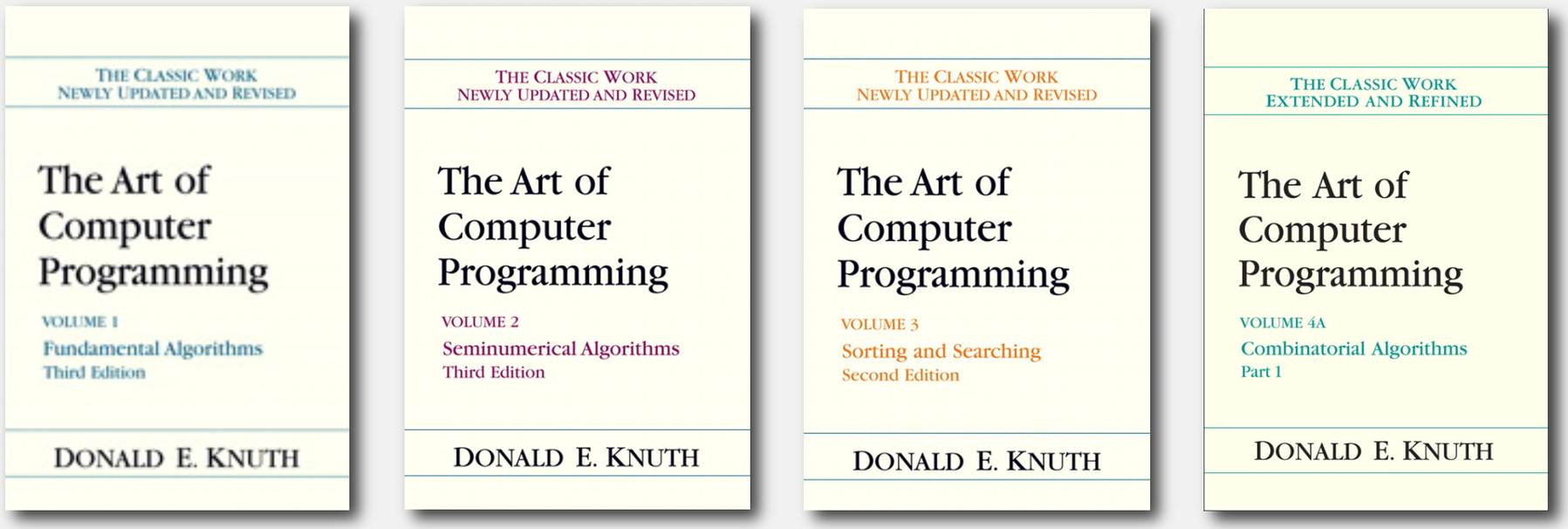
e.g., can run huge number of experiments

- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory

Mathematical models for running time

Total running time: sum of cost · 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	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

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

Cost of basic operations

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

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 2 N

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++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;=\; \frac{1}{2} N (N-1)$$

$$= \binom{N}{2}$$

operation	frequency
variable declaration	N + 2
assignment statement	N + 2
less than compare	½ (N + 1) (N + 2)
equal to compare	½ N (N − 1)
array access	N (N − 1)
increment	½ N (N − 1) to N (N − 1)

tedious to count exactly

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++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;=\; \frac{1}{2} N (N-1)$$

$$= \binom{N}{2}$$

operation	frequency
variable declaration	N + 2
assignment statement	N + 2
less than compare	½ (N + 1) (N + 2)
equal to compare	½ N (N − 1)
array access	N (N − 1)
increment	½ N (N − 1) to N (N − 1)

← cost model = array accesses

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

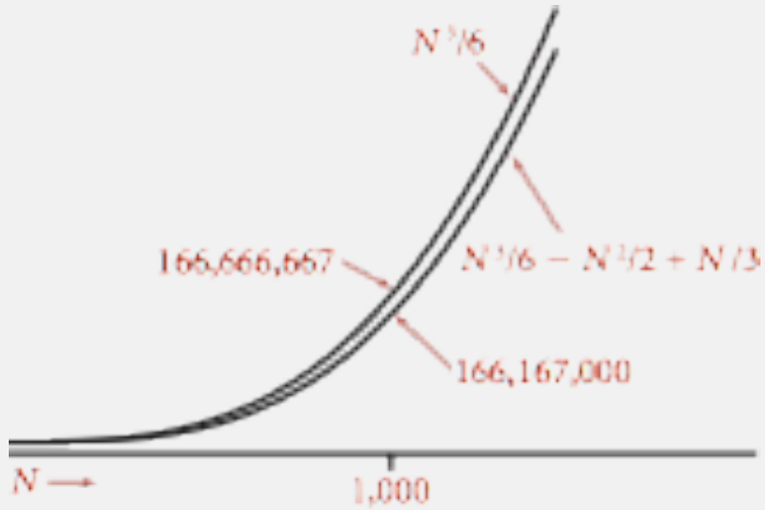
Ex 1. $\frac{1}{6} N^3 + 20 N + 16 \sim \frac{1}{6} N^3$

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

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)



Leading-term approximation

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

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	~ N
assignment statement	N + 2	~ N
less than compare	½ (N + 1) (N + 2)	~ ½ N^2
equal to compare	½ N (N − 1)	~ ½ N^2
array access	N (N − 1)	~ N^2
increment	½ N (N − 1) to N (N − 1)	~ ½ N^2 to ~ 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++;
```

"inner loop"

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

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

Bottom line. Use cost model and tilde notation to simplify frequency 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)
            count++;
```

"inner loop"

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 frequency counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

R. *A1.* Replace the sum with an integral, and use calculus!

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

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

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

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

Ex 3. 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.

costs (depend on machine, compiler)

$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$

A = array access
B = integer add
C = integer compare
D = increment
E = variable assignment

frequencies
(depend on algorithm, input)

Bottom line. Use approximate models : $T(N) \sim c\, N^3$.

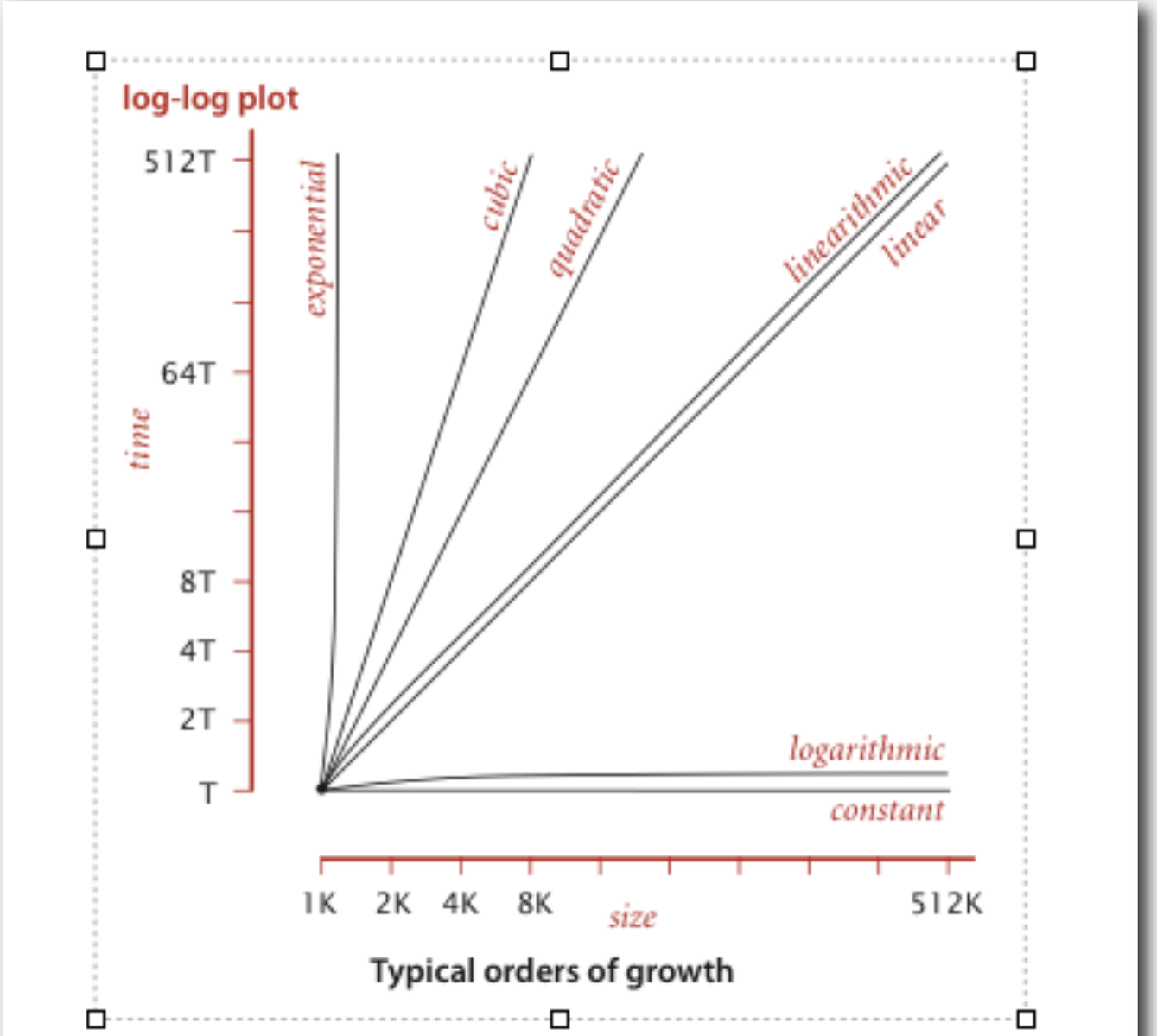
- observations
- mathematical models
- **order-of-growth classifications**
- dependencies on inputs
- memory

Common order-of-growth classifications

Good news. the small set of functions

$$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3, \text{ and } 2^N$$

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



Common order-of-growth classifications

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

Q. How many inputs can be processed in minutes?

Ex. Customers lost patience waiting "minutes" in 1970s; they still do.

Q. How long to process millions of inputs?

Ex. Population of NYC was "millions" in 1970s; still is.

seconds	equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
...	forever
10^{17}	age of universe

For back-of-envelope calculations, assume:

decade	processor speed	instructions per second
1970s	1 MHz	10^6
1980s	10 MHz	10^7
1990s	100 MHz	10^8
2000s	1 GHz	10^9

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.

Practical implications of order-of-growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
N log N	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millennia

Practical implications of order-of-growth

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	-	-
log N	logarithmic	nearly independent of input size	-	-
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4-5x
2^N	exponential	useful only for tiny problems	forever	1x

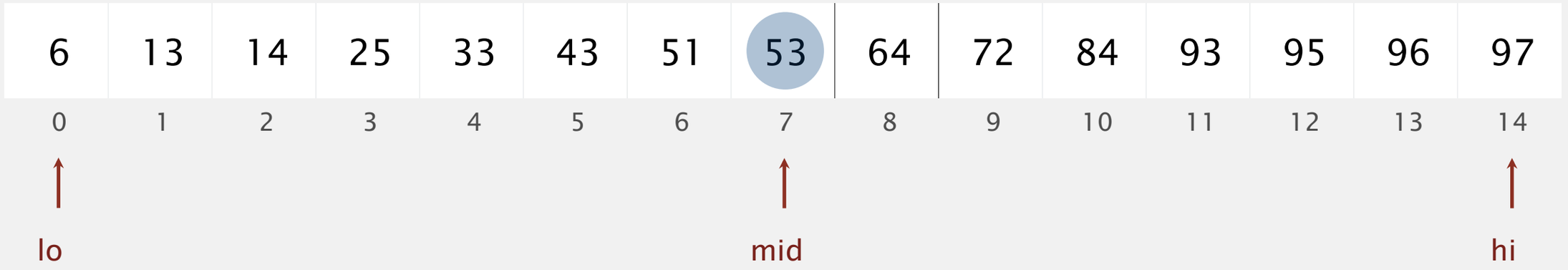
Binary search

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.

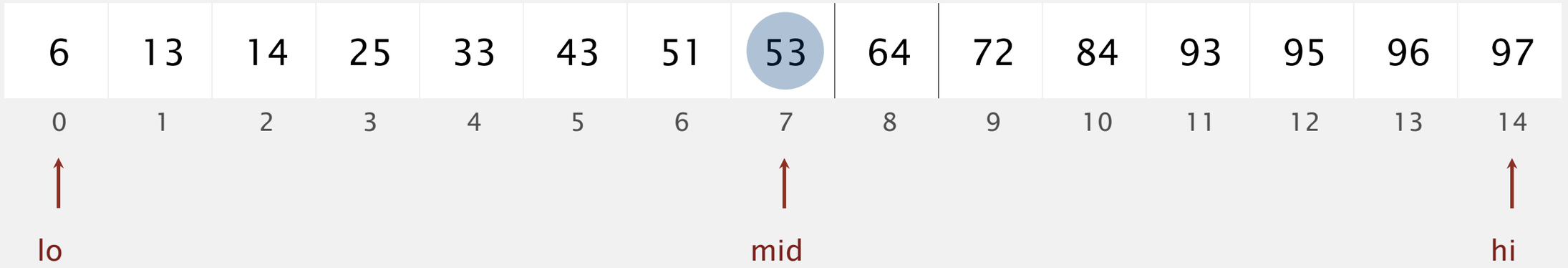
recursion



Binary search

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

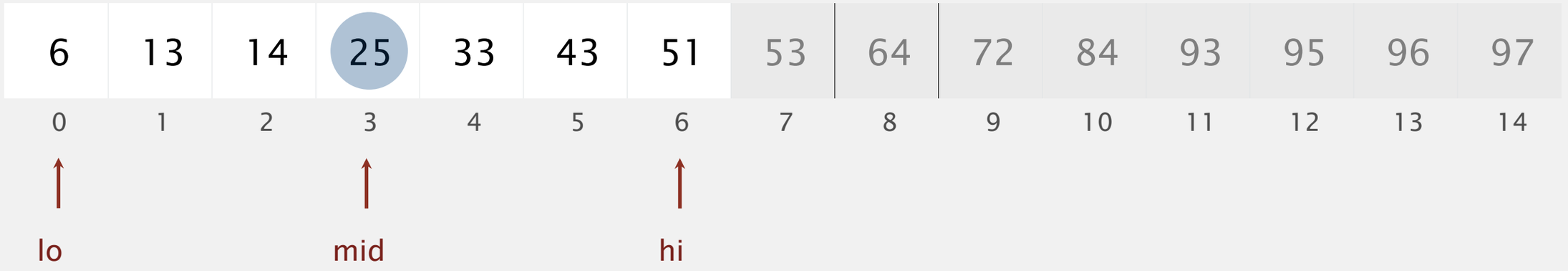
Successful search. Binary search for 33.



Binary search

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

Successful search. Binary search for 33.



Binary search

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

Successful search. Binary search for 33.

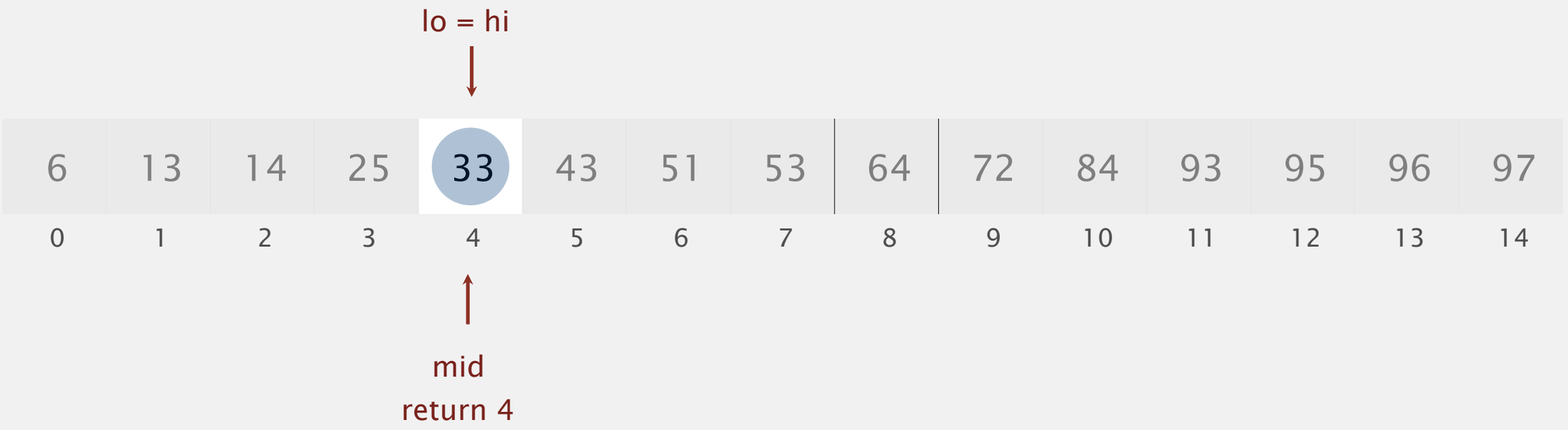
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

lo mid hi

Binary search

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

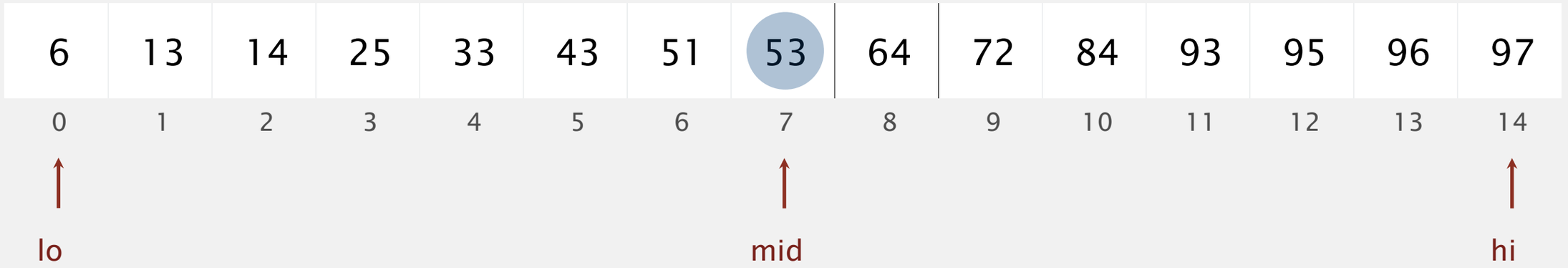
Successful search. Binary search for 33.



Binary search

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

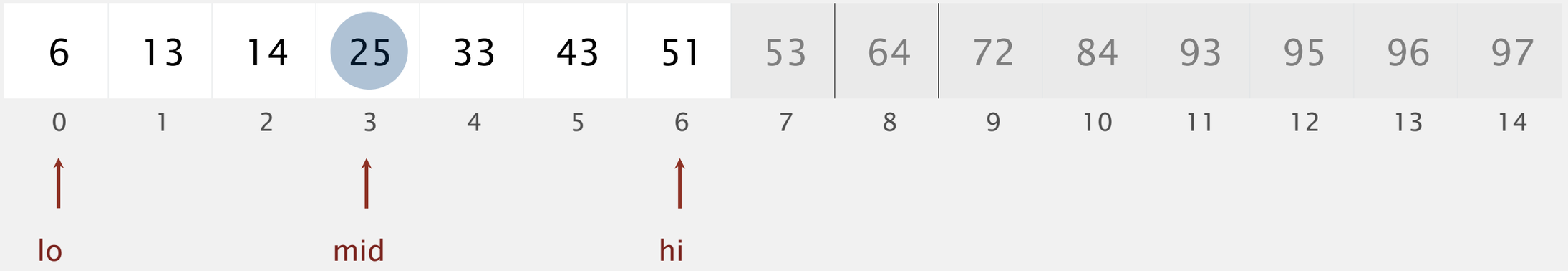
Unsuccessful search. Binary search for 34.



Binary search

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

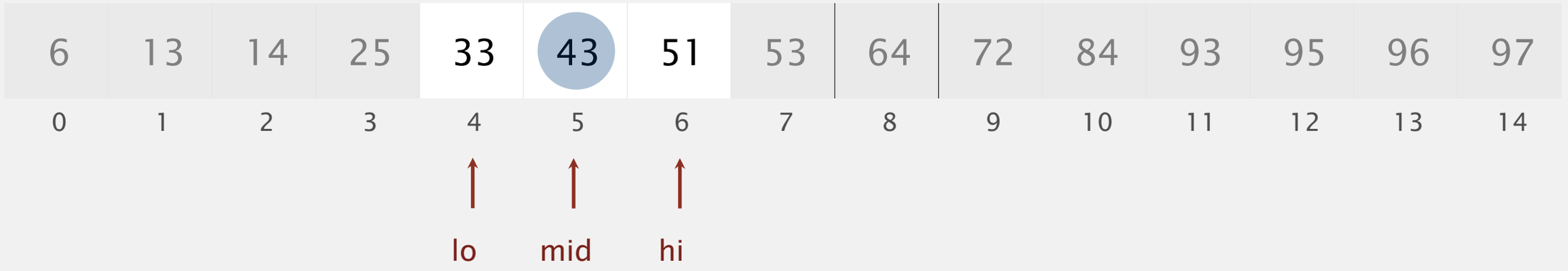
Unsuccessful search. Binary search for 34.



Binary search

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

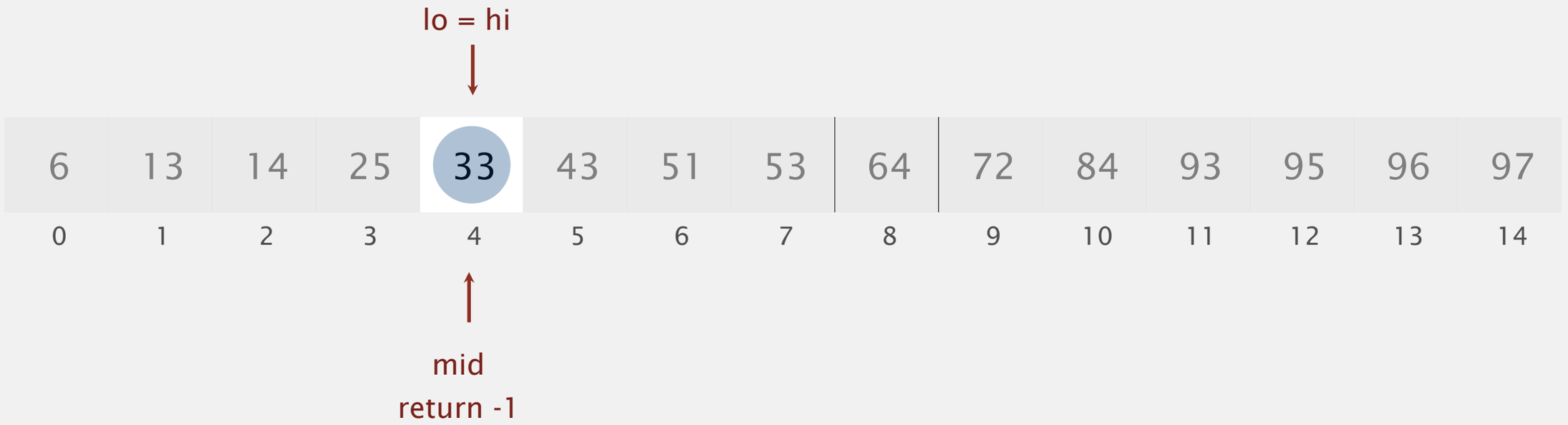
Unsuccessful search. Binary search for 34.



Binary search

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

Unsuccessful search. Binary search for 34.



Trace of binary search

			a[]												
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo	hi	mid															
0	14	7	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	6	3	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
4	6	5	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
4	4	4	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

entries in black are `a[lo..hi]`

entry in red is `a[mid]`

loop exits with `a[mid] = 33: return 4`

Trace of successful binary search for 33

			a[]												
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo	hi	mid															
0	14	7	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
8	14	11	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
8	10	9	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
8	8	8	6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
9	8		6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

loop exits with `lo > hi: return -1`

Trace of unsuccessful binary search for 65

Binary search: mathematical analysis

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

Def. $T(N)$ ≡ # compares to binary search in a sorted subarray of size at most N.

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

Pf sketch.

↑ left or right half

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

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

Binary search: mathematical analysis

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

Def. $T(N)$ ≡ # compares to binary search in a sorted subarray of size at most N.

Binary search recurrence. $T(N) \leq T(\lfloor N / 2 \rfloor) + 1$ for $N > 1$, with $T(0) = 0$.

For simplicity, we prove when $N = 2^n - 1$ for some n, so $\lfloor N / 2 \rfloor = 2^{n-1} - 1$.

$$
\begin{aligned}
T(2^n - 1) &\leq T(2^{n-1} - 1) + 1 && \text{given} \\
&\leq T(2^{n-2} - 1) + 1 + 1 && \text{apply recurrence to first term} \\
&\leq T(2^{n-3} - 1) + 1 + 1 + 1 && \text{apply recurrence to first term} \\
&\cdots \\
&\leq T(2^0 - 1) + 1 + 1 + \ldots + 1 && \\
&= n && \text{stop applying, T(0) = 1}
\end{aligned}
$$

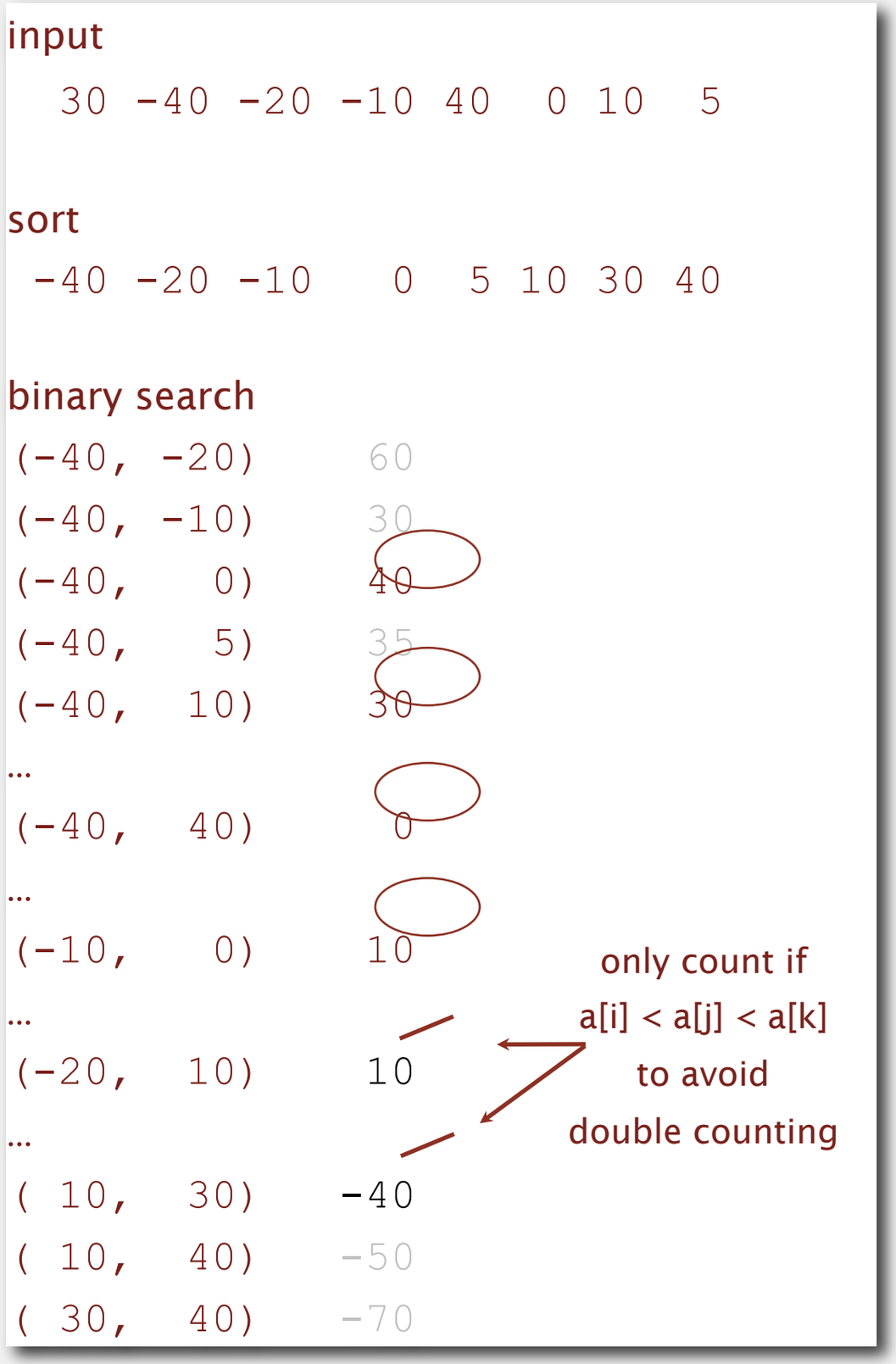
An N^2 log N algorithm for 3-sum

Algorithm.

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

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 $N^2 \log N$ three-sum algorithm is significantly faster in practice than the brute-force N^3 one.

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

Bottom line. Typically, better order of growth $\Rightarrow$ faster in practice.

- observations
- mathematical models
- order-of-growth classifications
- **dependencies on inputs**
- 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. Compares 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.

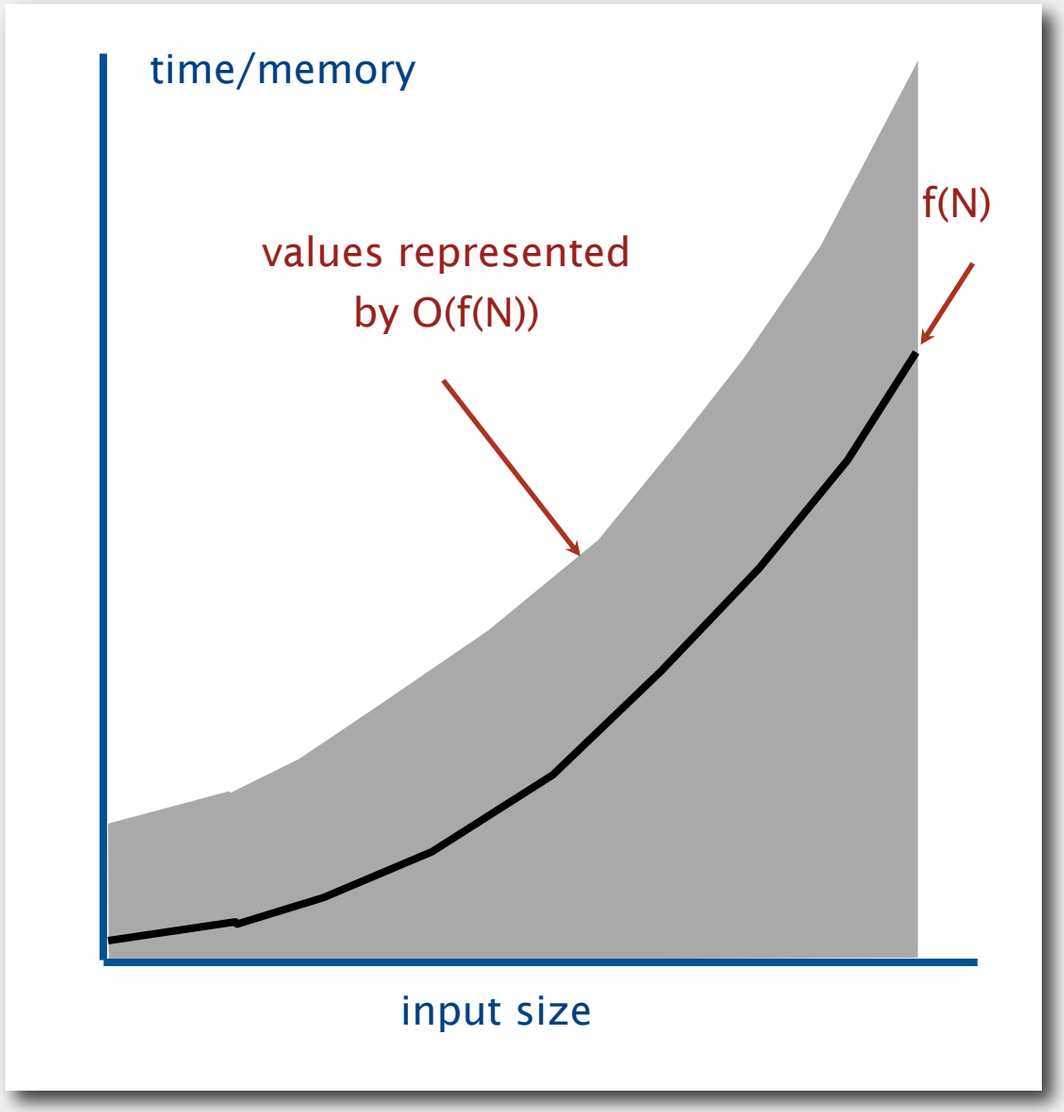
Commonly-used notations

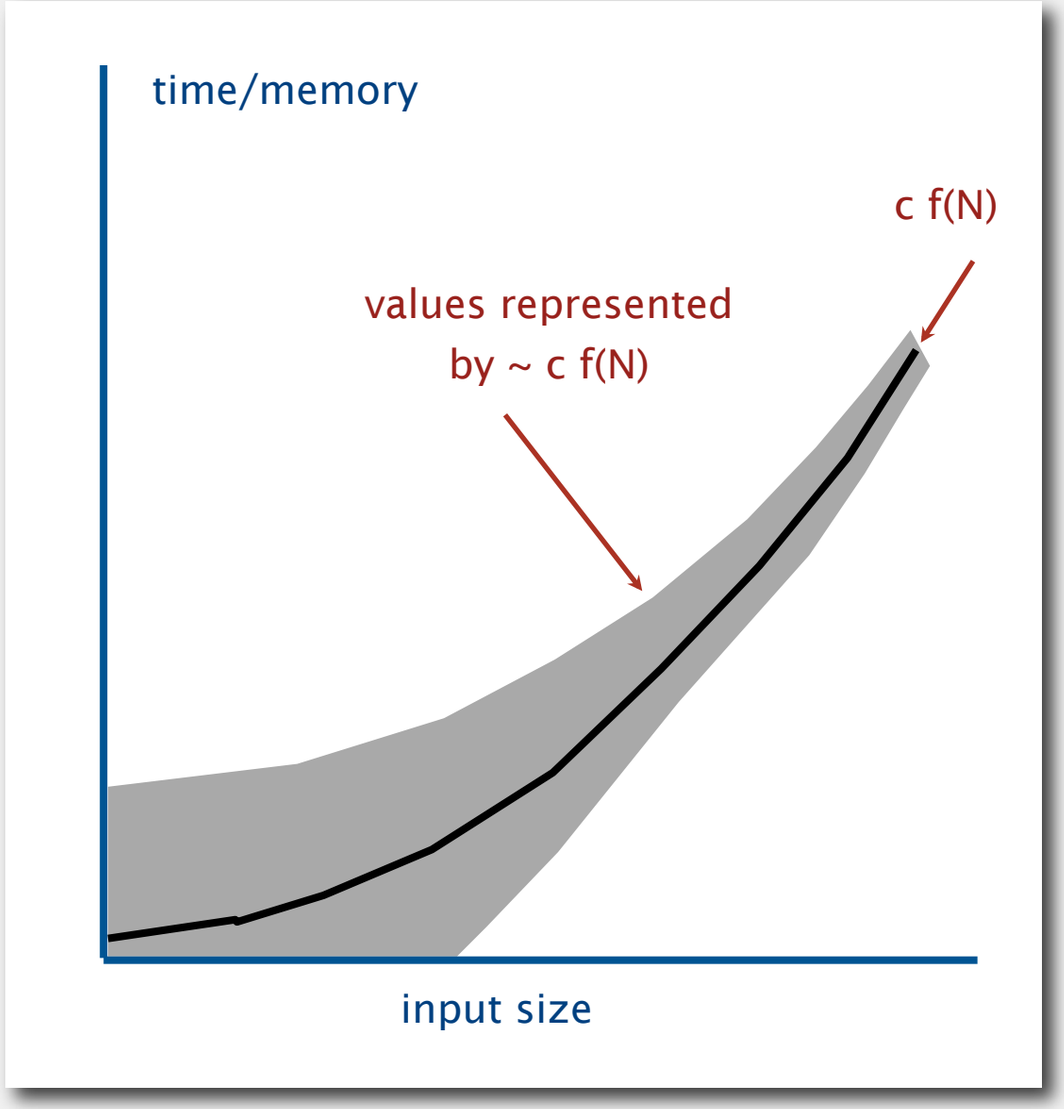
notation	provides	example	shorthand for	used to
Tilde	leading term	~ 10 N^2	10 N^2 10 N^2 + 22 N log N 10 N^2 + 2 N + 37	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	½ N^2 10 N^2 5 N^2 + 22 N log N + 3N	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	10 N^2 100 N 22 N log N + 3 N	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	½ N^2 N^5 N^3 + 22 N log N + 3 N	develop lower bounds

Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.

- Big-Oh notation suppresses leading constant.
- Big-Oh notation only provides upper bound (not lower bound).
- Big-Omega is used to describe a lower bound on the worst case
- Big-Theta is used to describe performance of algo that are optimal, i.e., in the sense that no algo can have better asymptotic worse-case growth





Examples

Ex 1. Our brute-force 3-sum algorithm takes $\Theta(N^3)$ time.

Ex 2. Conjecture: worst-case running time for any 3-sum algorithm is $\Omega(N^2)$.

Ex 3. Insertion sort uses $O(N^2)$ compares to sort any array of N elements; it uses $\sim N$ compares in the best case (already sorted) and $\sim \frac{1}{2}N^2$ compares in the worst case (reverse sorted).

Ex 4. The worst-case height of a tree created with union find with path compression is $\Theta(N)$.

Ex 5. The height of a tree created with weighted quick union is $O(\log N)$.

base of logarithm absorbed by big-Oh

$$\log_a N = \frac{1}{\log_b a} \log_b N$$

- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- **memory**

Basics

Bit. 0 or 1.
Byte. 8 bits.
Megabyte (MB). 1 million bytes.
Gigabyte (GB). 1 billion bytes.

Old machine. We used to assume a 32-bit machine with 4 byte pointers.

Modern 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

Array overhead. 24 bytes.

.

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[][]	~ 2 M N
int[][]	~ 4 M N
double[][]	~ 8 M N

for two-dimensional arrays

Typical memory usage for objects in Java

Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Objects use 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;
...
}
```

16 bytes (object overhead)

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding)

32 bytes

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 if inner class.

padding: round up to multiple of 8

extra pointer to enclosing class

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.

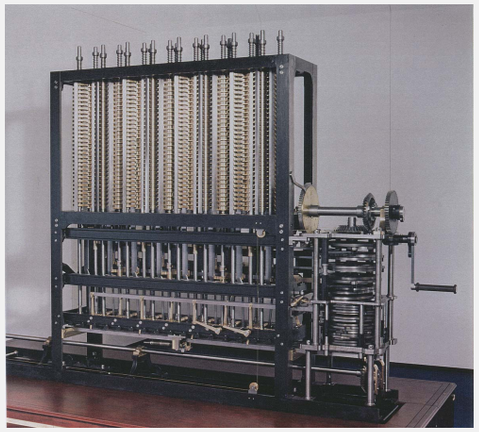
Turning the crank: summary

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.

Art of Algorithms

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

Analyzing and improving Algorithm used

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

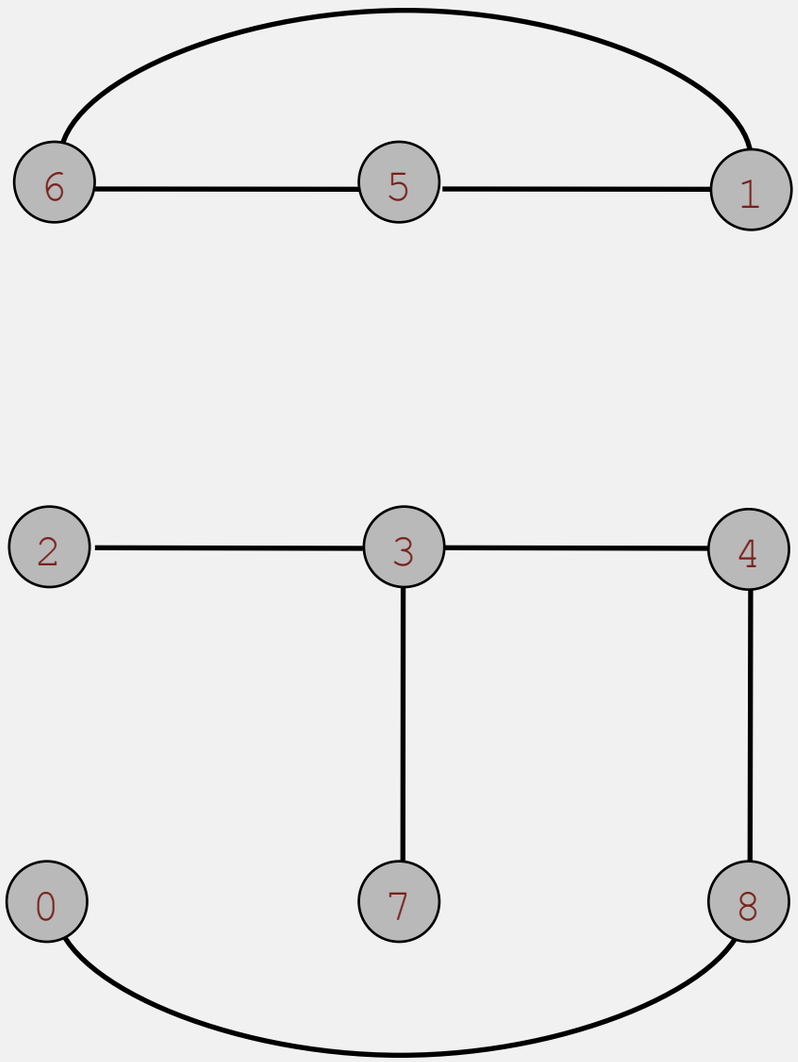
Dynamic connectivity

Given a set of objects

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

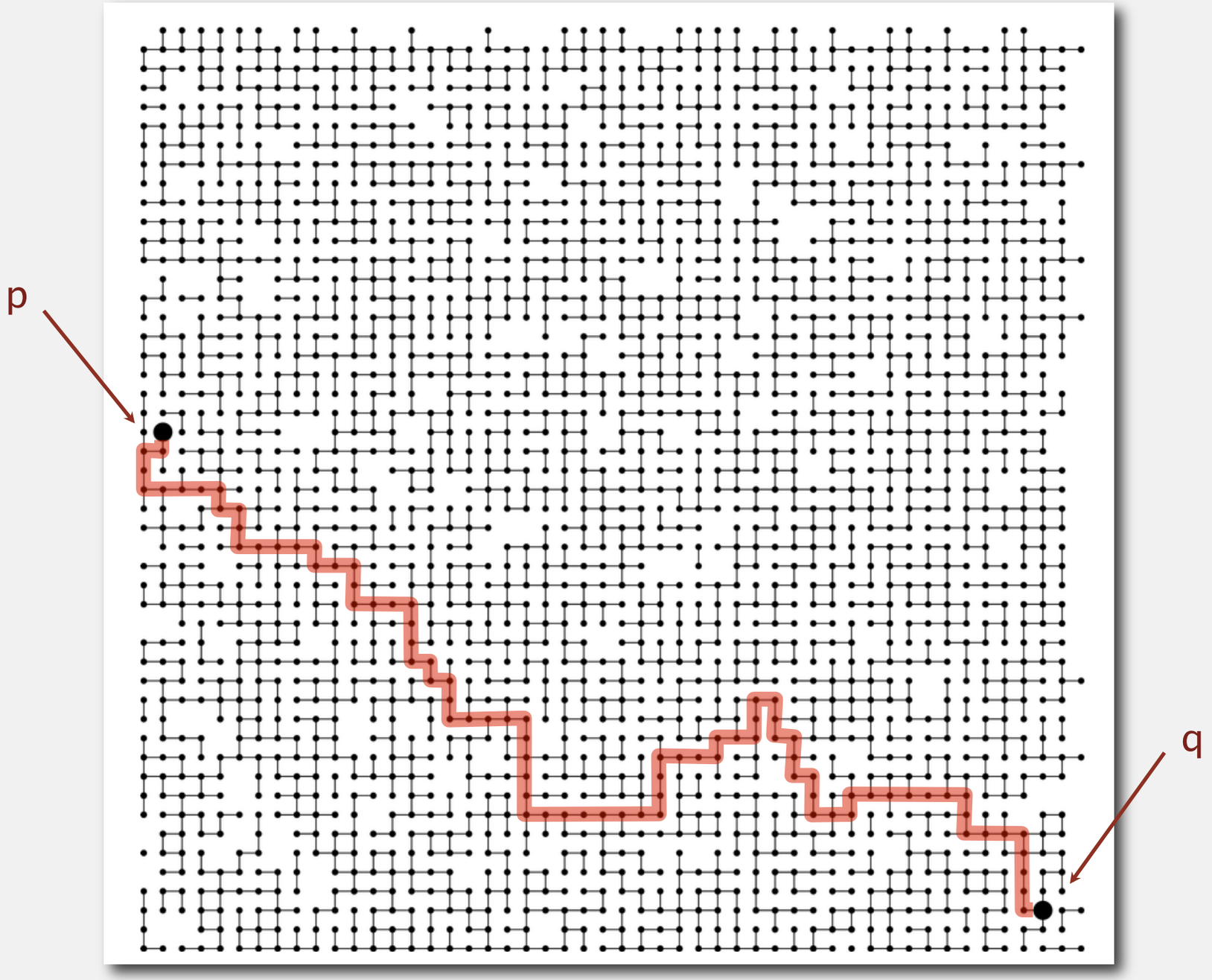
more difficult problem: find the path

```
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
```



Connectivity example

Q. Is there a path from p to q?



A. Yes.

Modeling the objects

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.

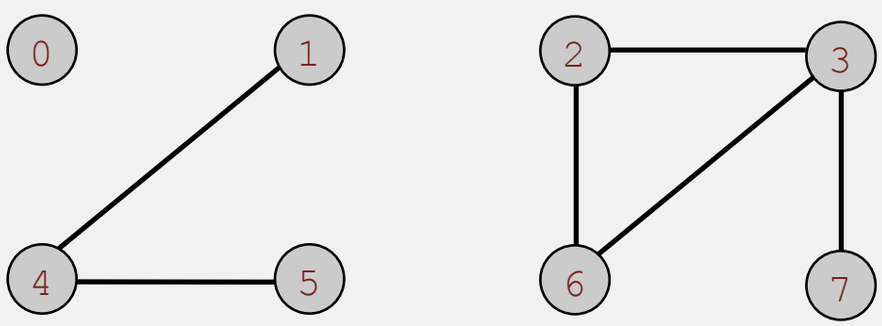
can use symbol table to translate from site names to integers: stay tuned (Chapter 3)

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 }

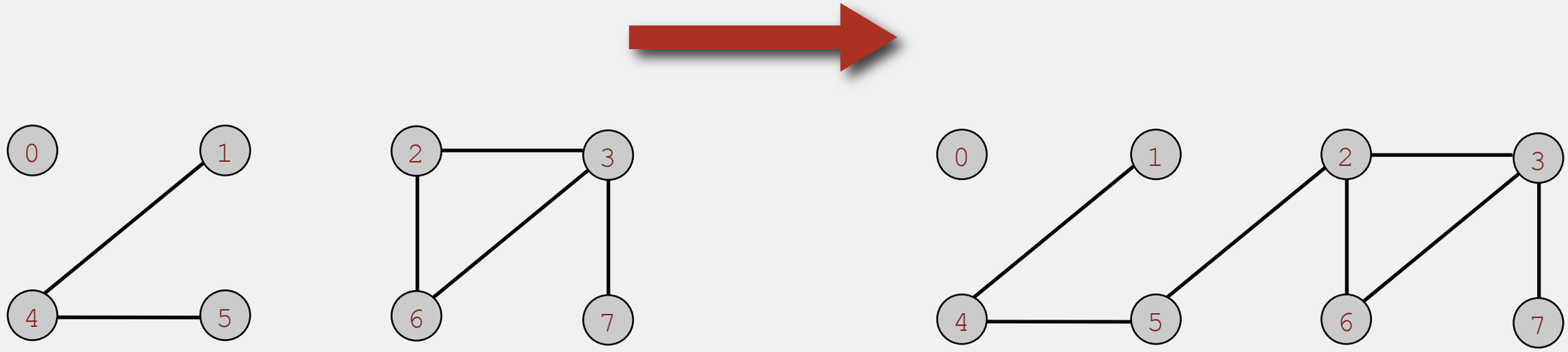
3 connected components

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)
```



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
```

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

Quick-find [eager approach]

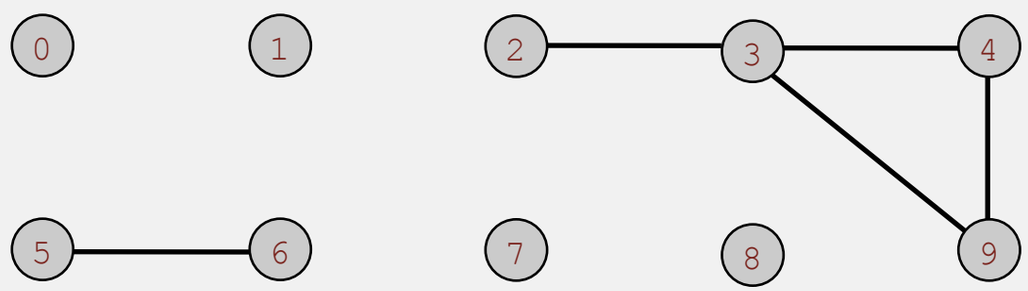
Data structure.

- 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



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

problem: many values can change

after union of 3 and 6

Quick-find example

		id[]									
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[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

```
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]`

(at most 2N + 1 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.

Ex. Takes N^2 array accesses to process sequence of N union commands on N objects.

Quadratic algorithms do not scale

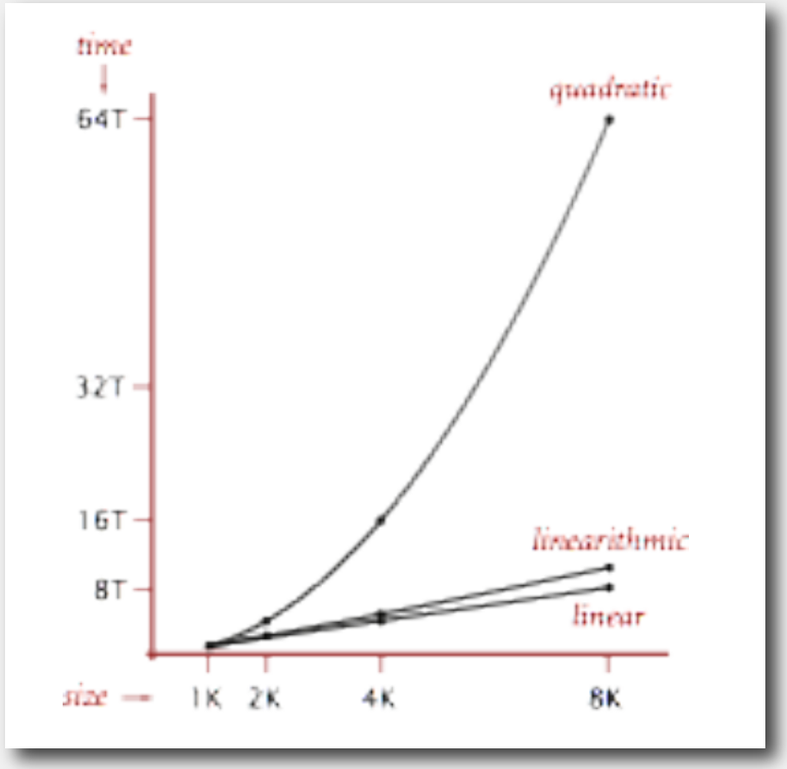
Rough standard (for now).

- 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!



Quadratic algorithms don't scale with technology.

- 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!

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

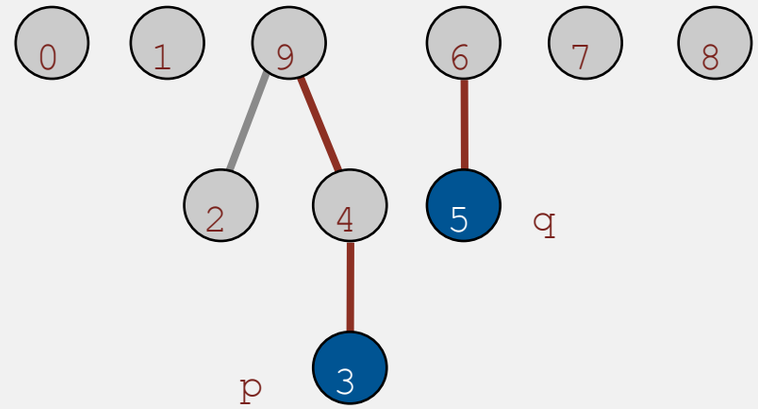
Quick-union [lazy approach]

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

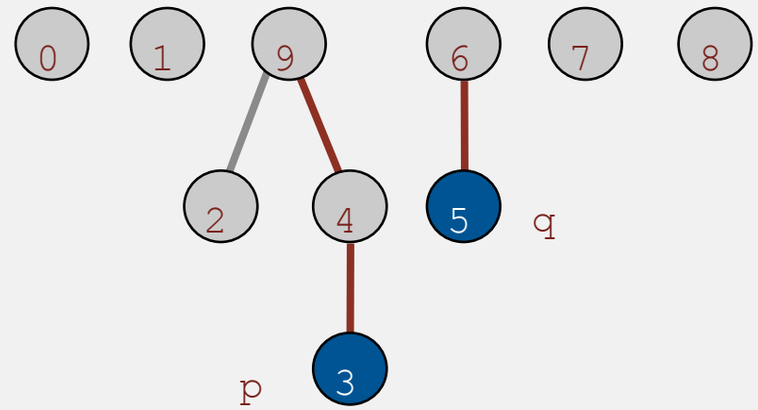
Quick-union [lazy approach]

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

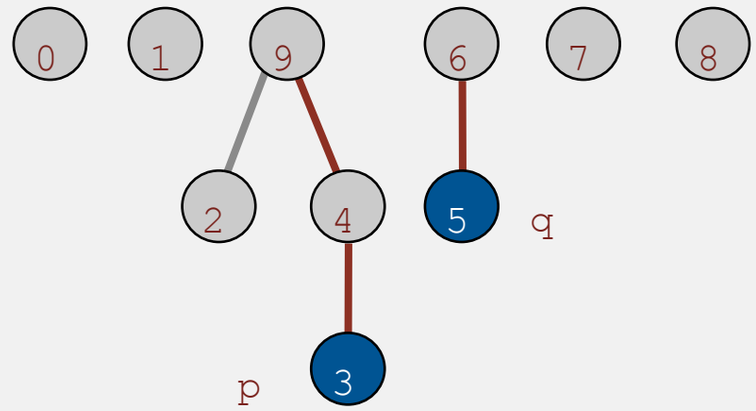
Quick-union [lazy approach]

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



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

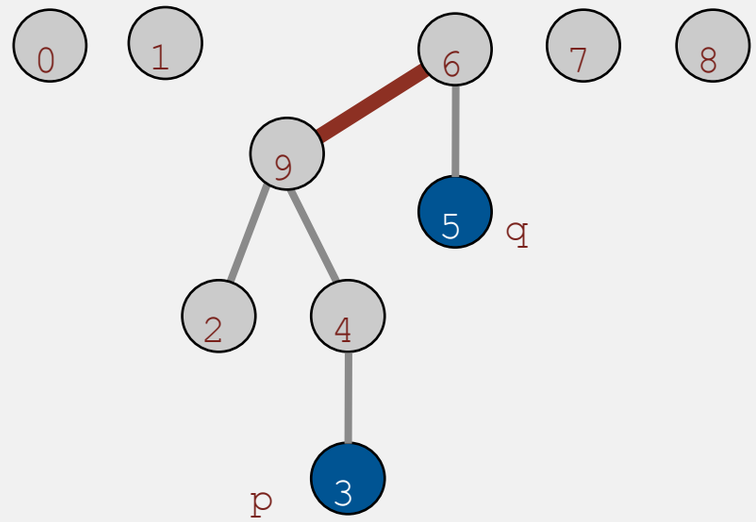
3 and 5 are not connected

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

Union. To merge components containing `p` and `q`, set the id of `p`'s root to the id of `q`'s root.

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



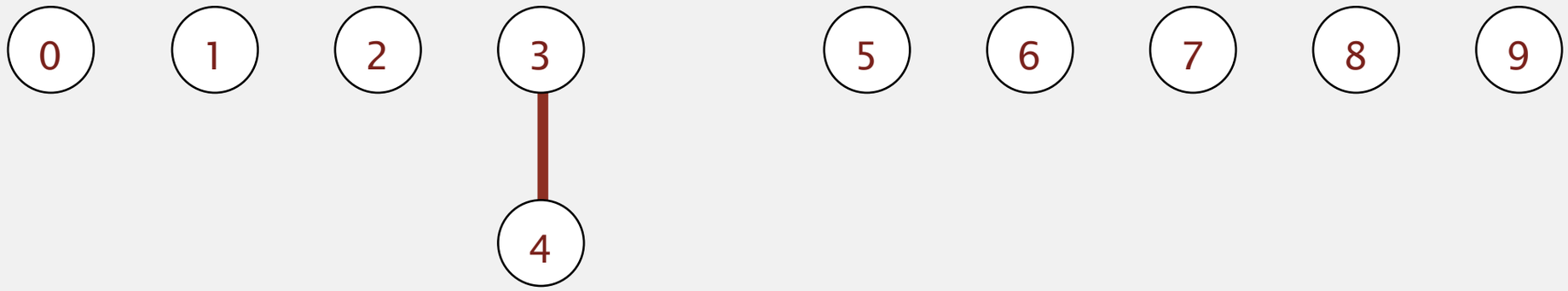
Quick-union demo

Quick-union demo

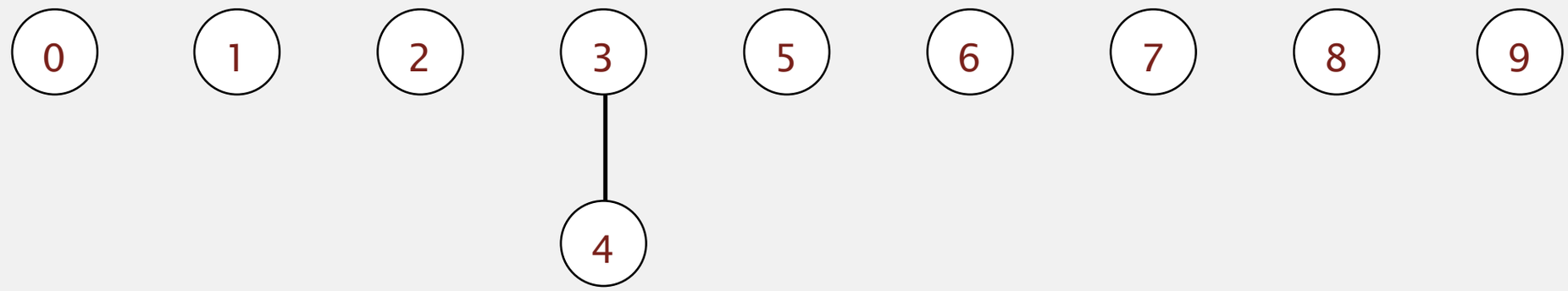
Union 4–3

Quick-union demo

Union 4–3

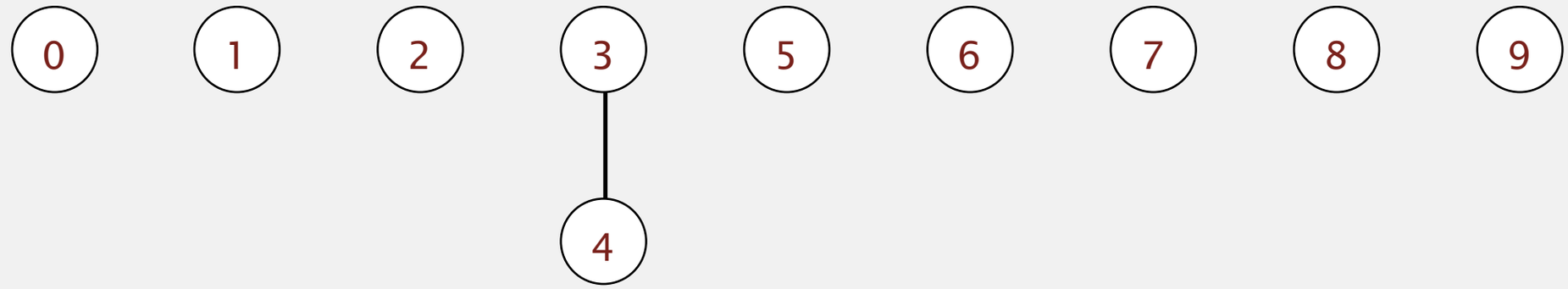


Quick-union demo



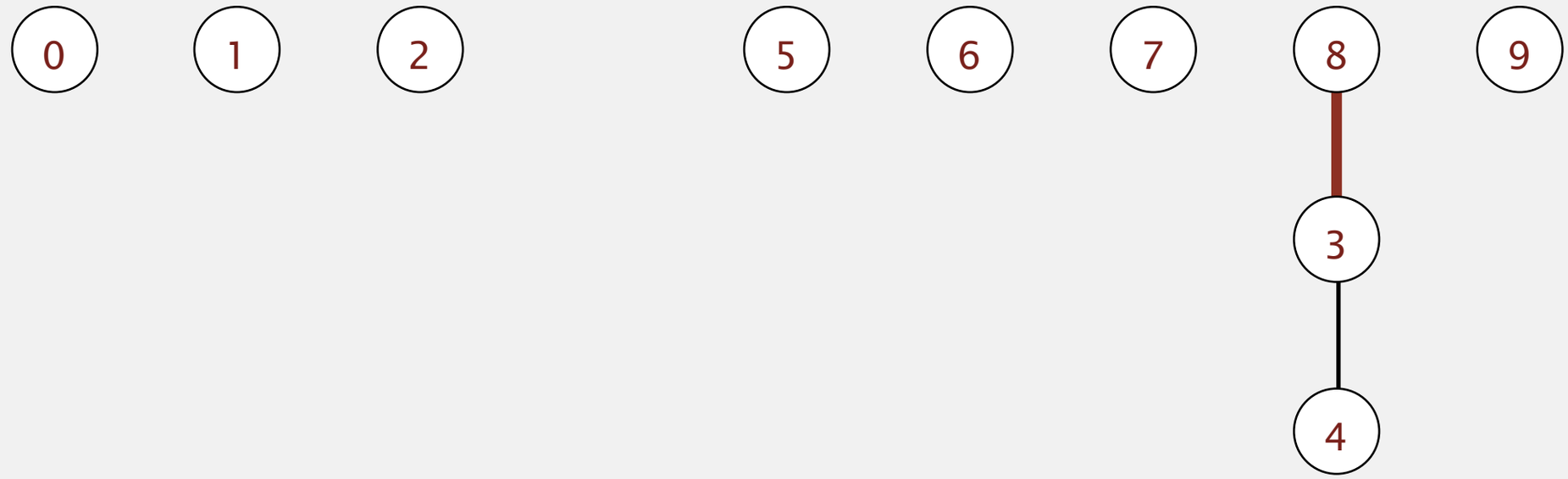
Quick-union demo

Union 3-8

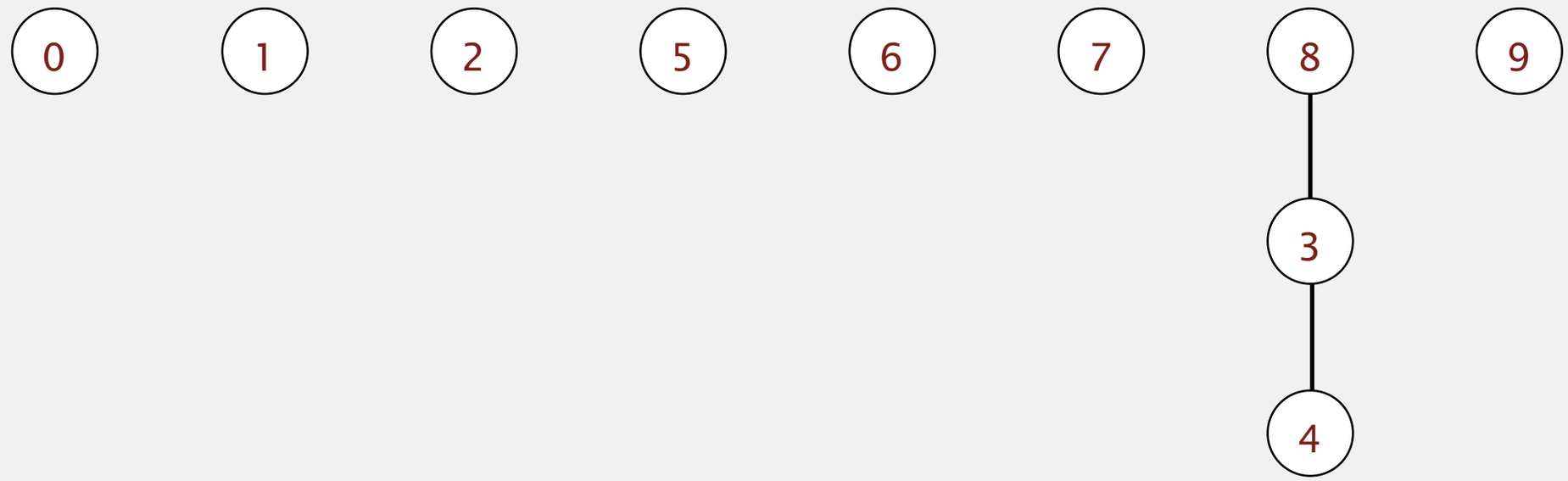


Quick-union demo

Union 3-8



Quick-union demo

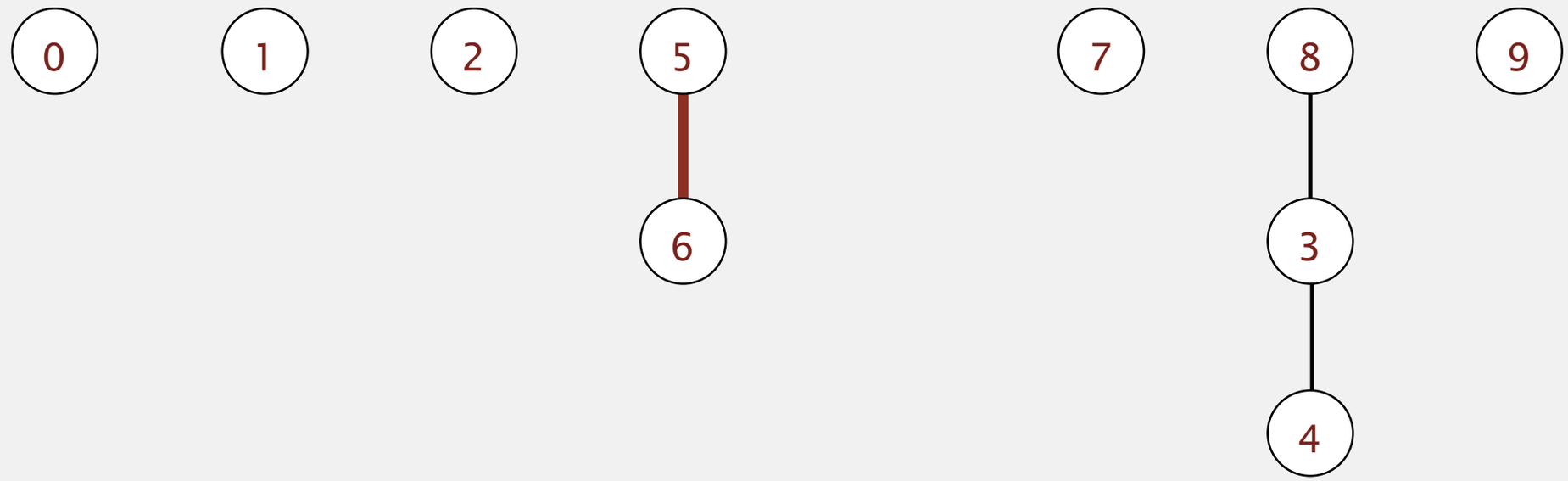


Quick-union demo

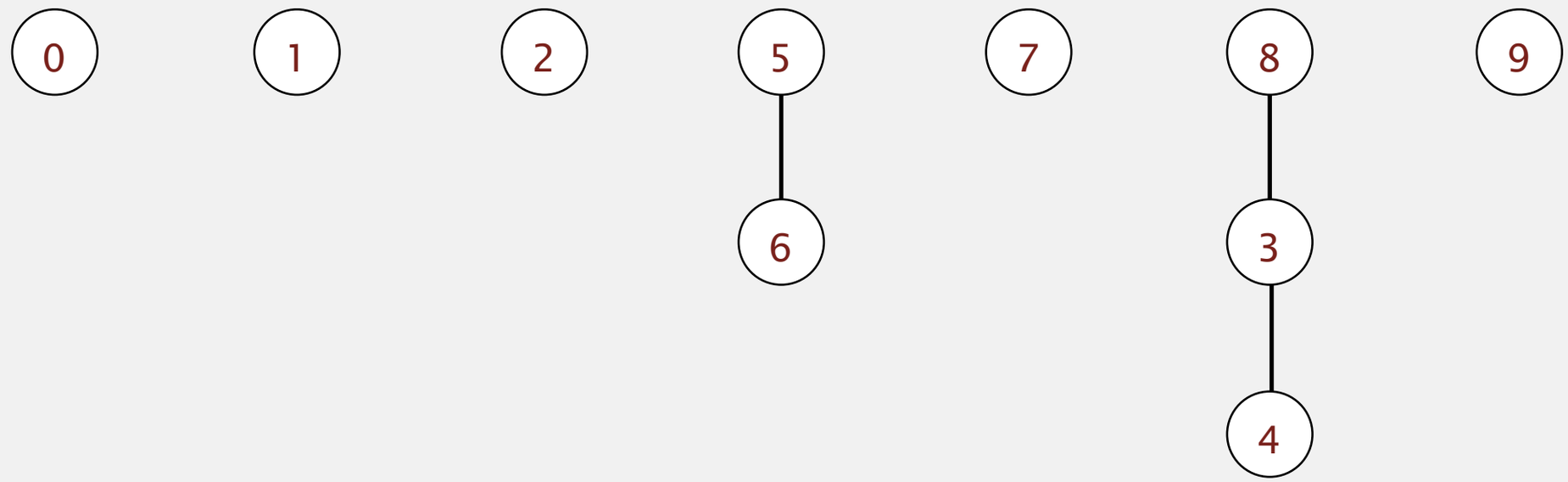
Union 6-5

Quick-union demo

Union 6-5

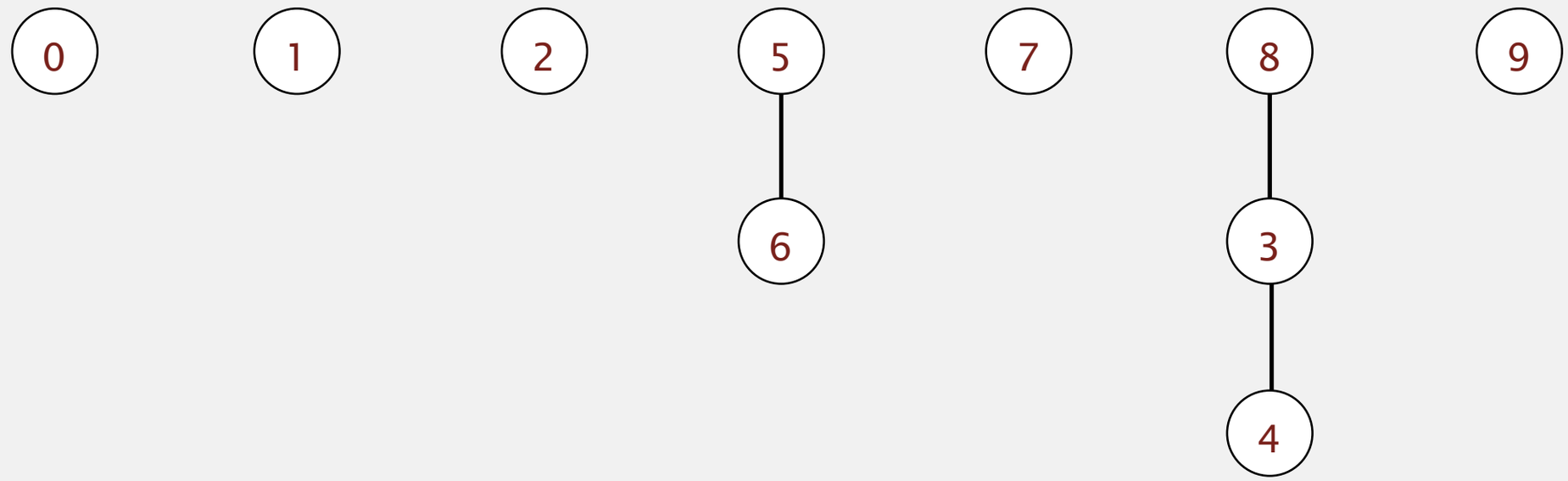


Quick-union demo



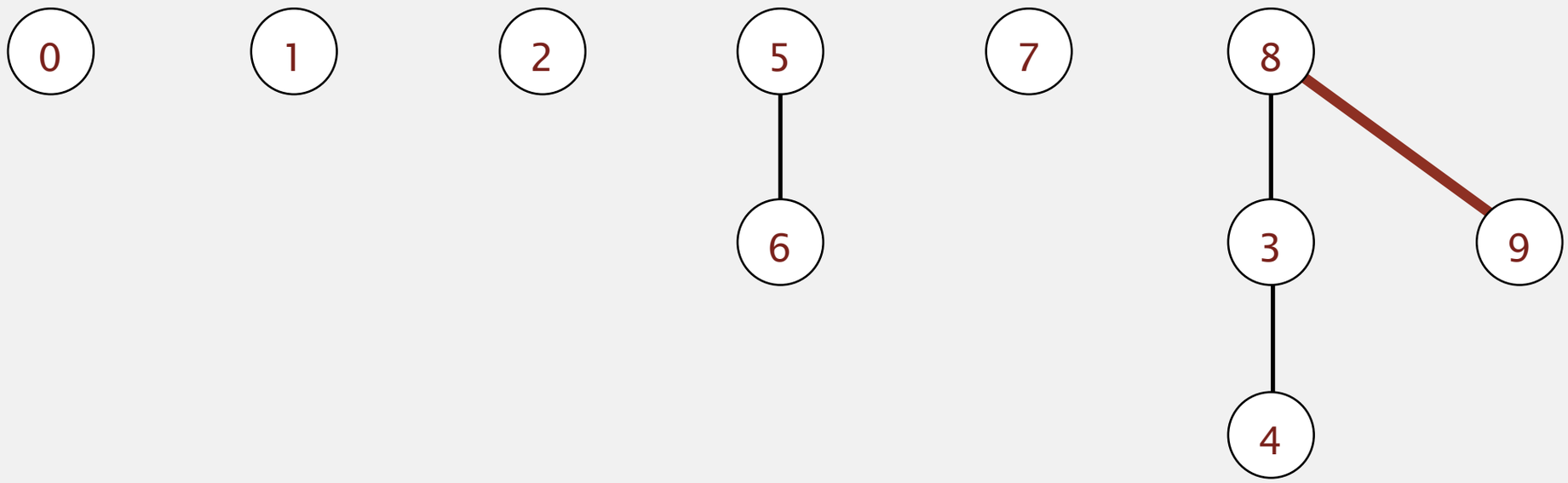
Quick-union demo

Union 9–4

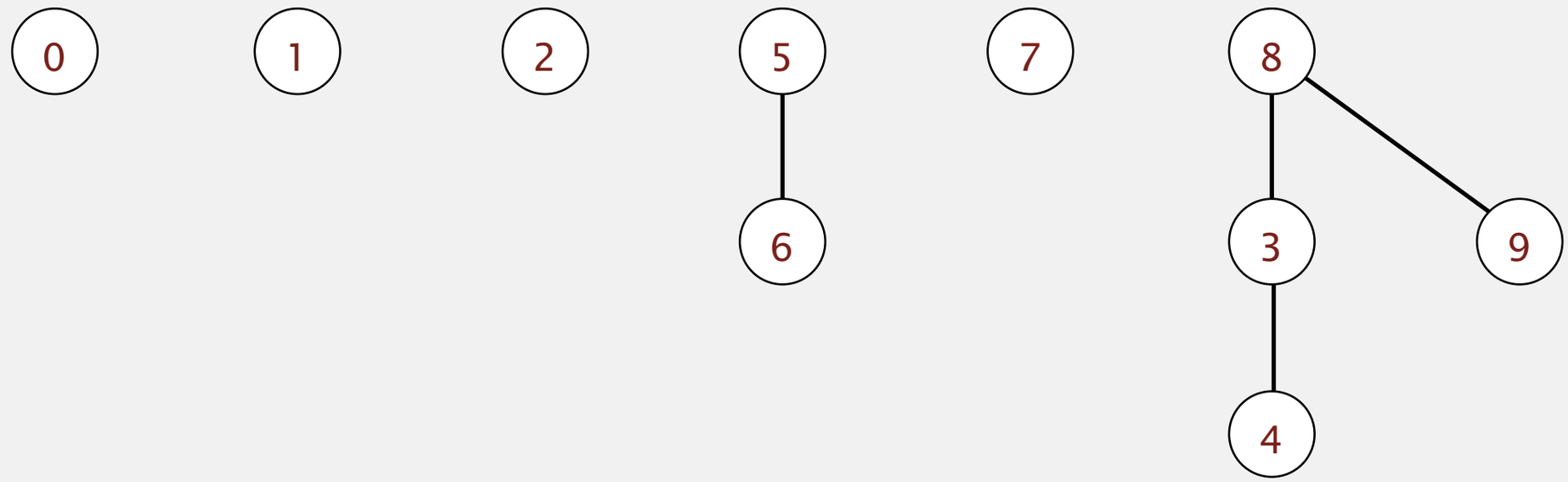


Quick-union demo

Union 9–4

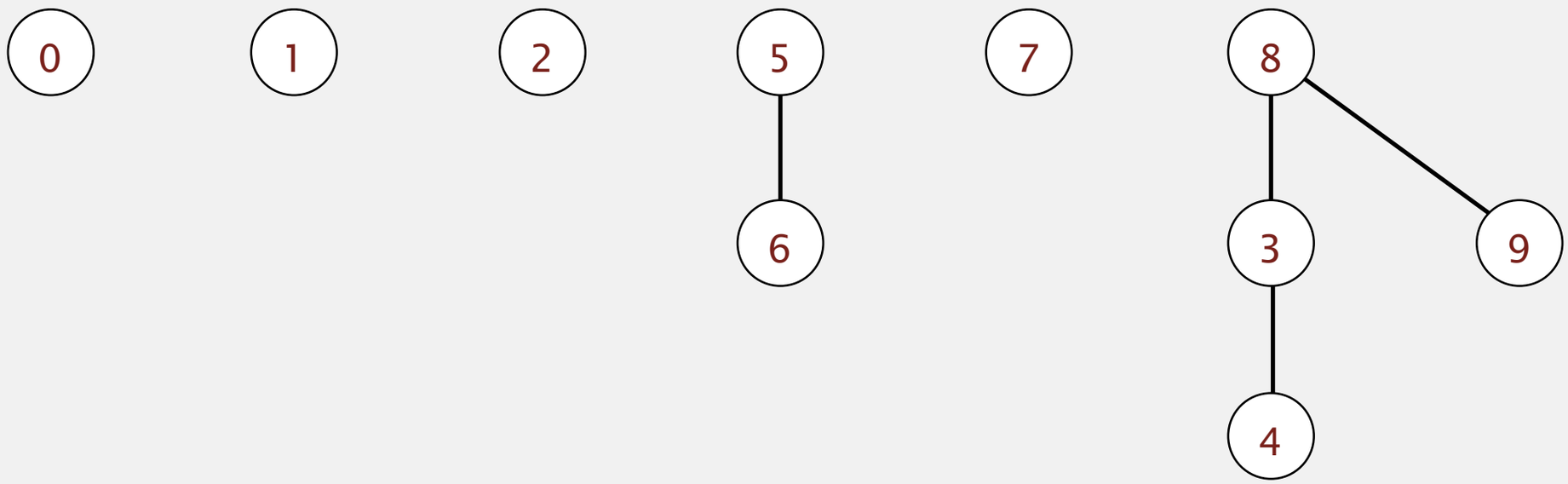


Quick-union demo



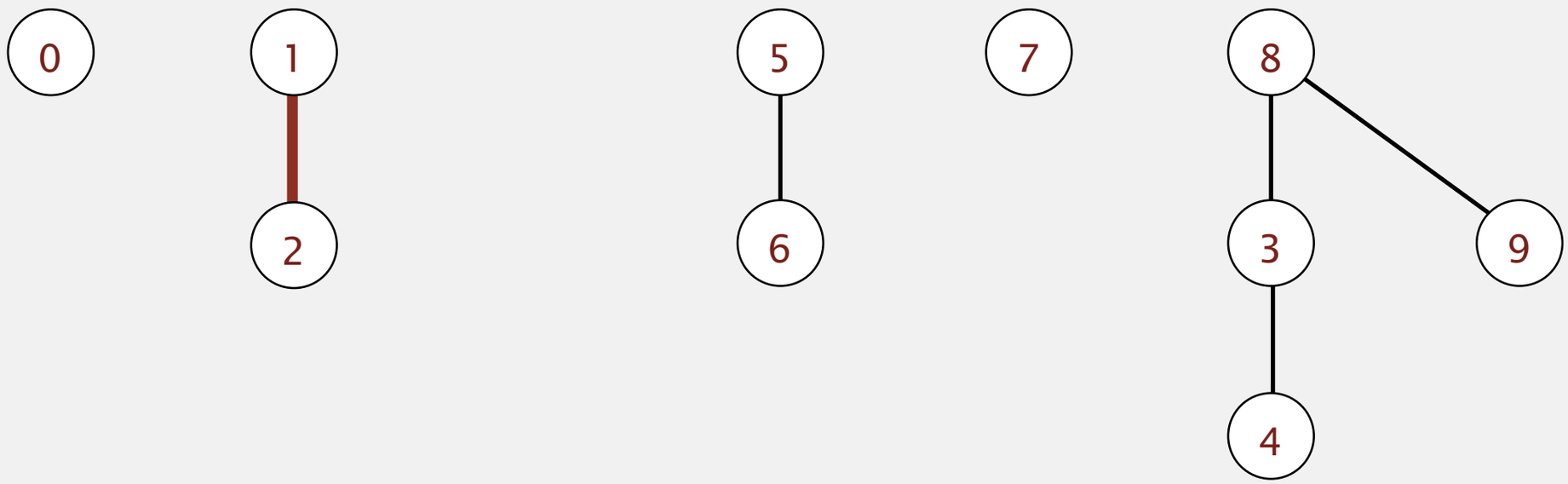
Quick-union demo

Union 2-1

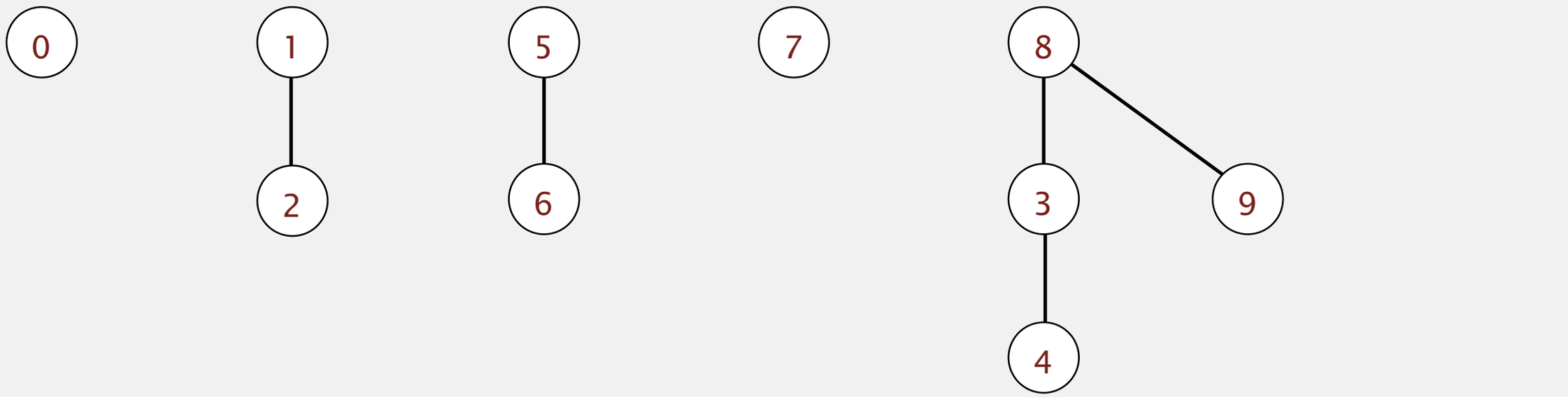


Quick-union demo

Union 2-1

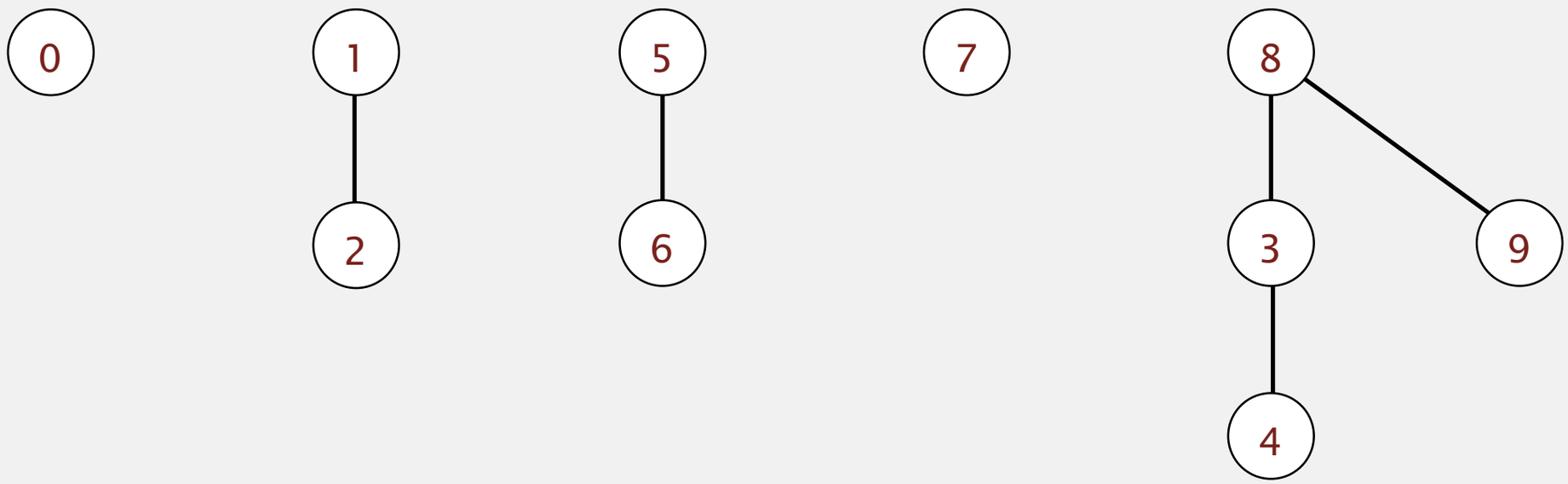


Quick-union demo



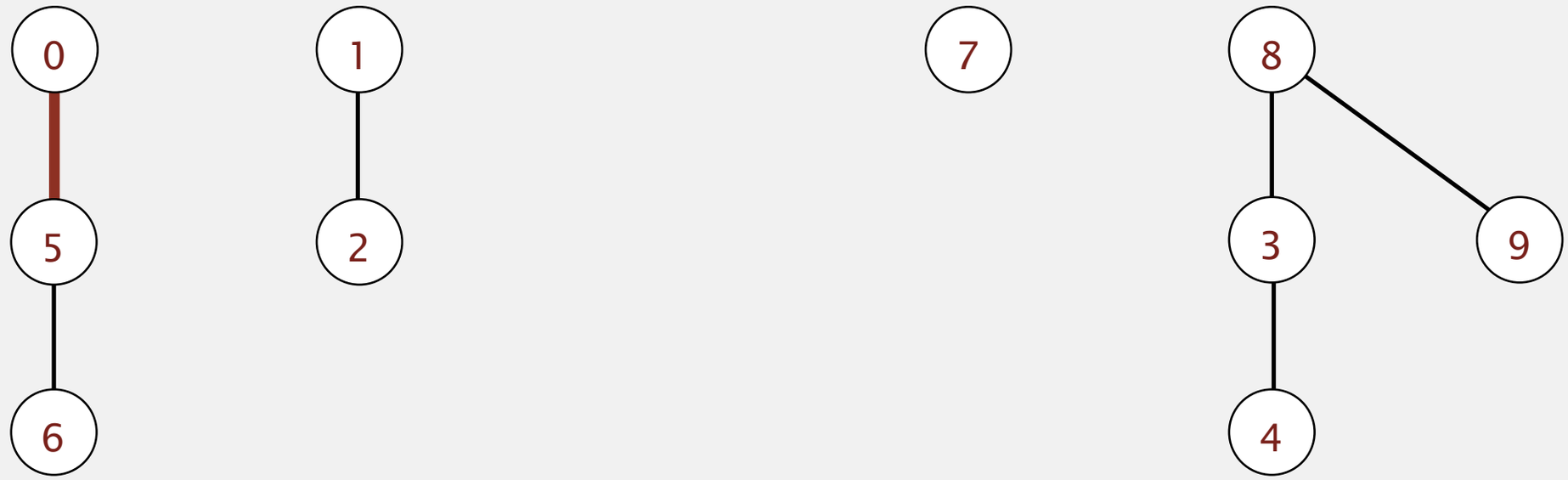
Quick-union demo

Union 5-0

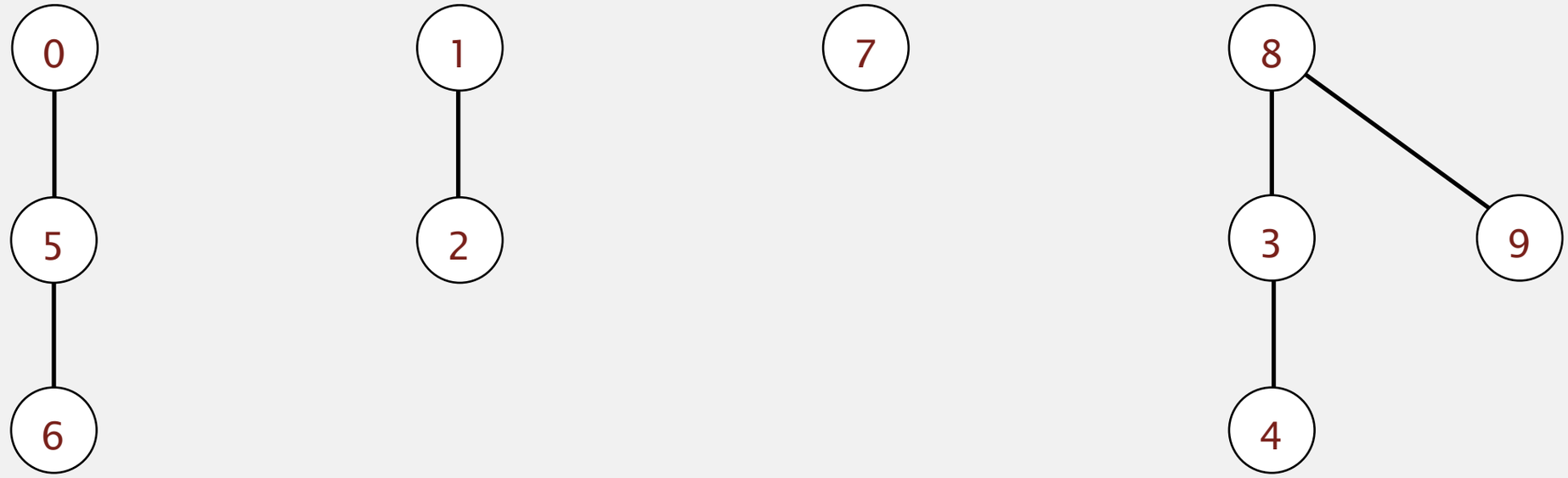


Quick-union demo

Union 5-0

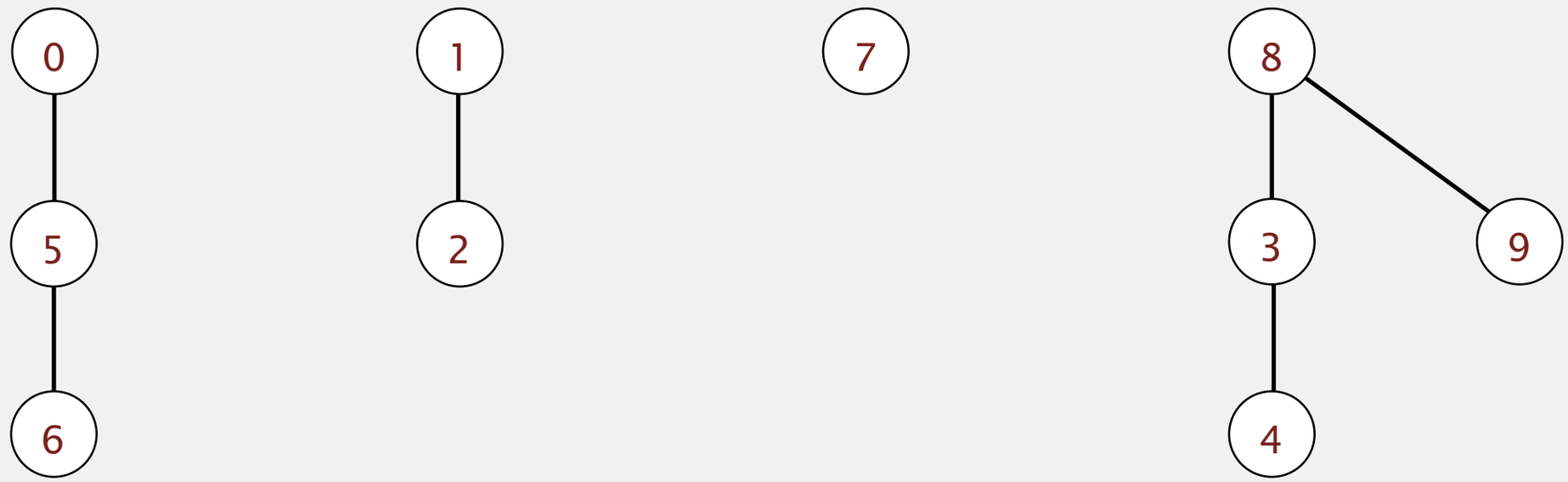


Quick-union demo



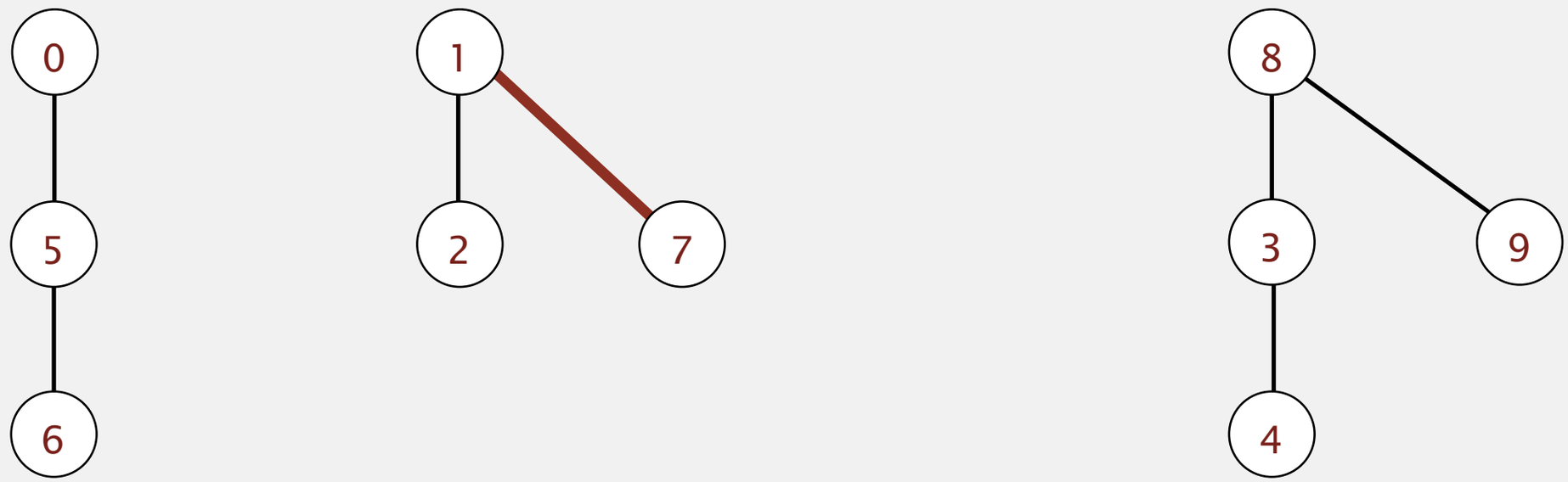
Quick-union demo

Union 7–2

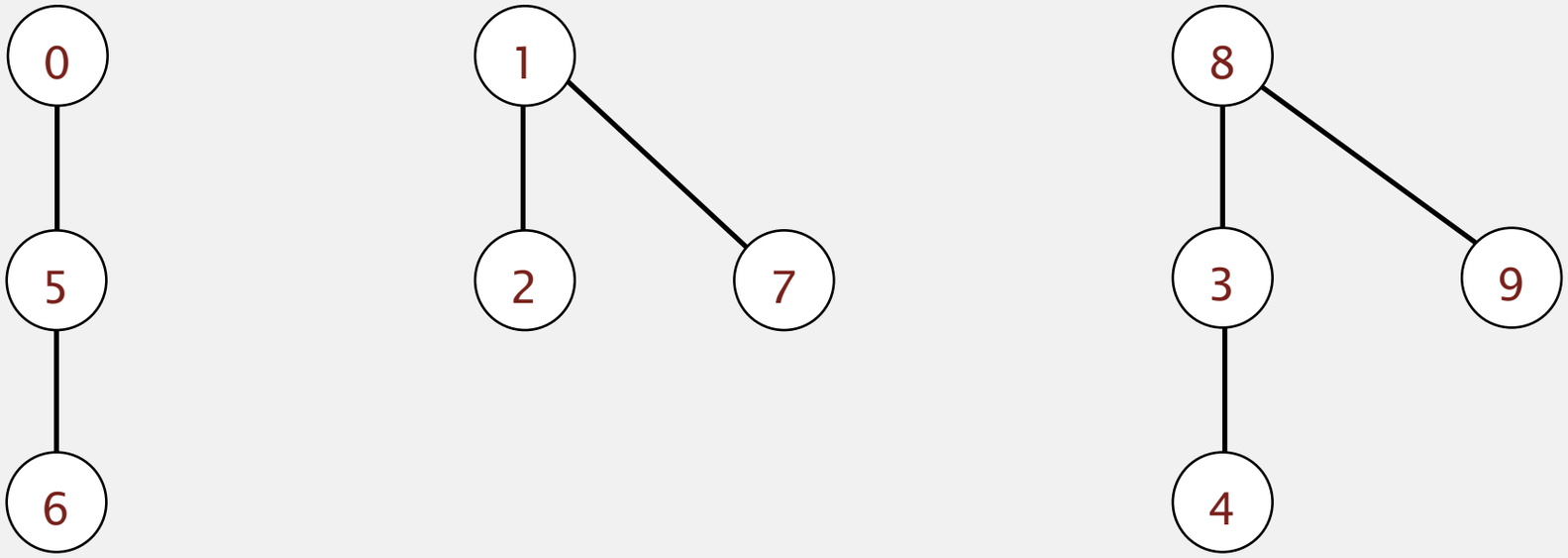


Quick-union demo

Union 7-2

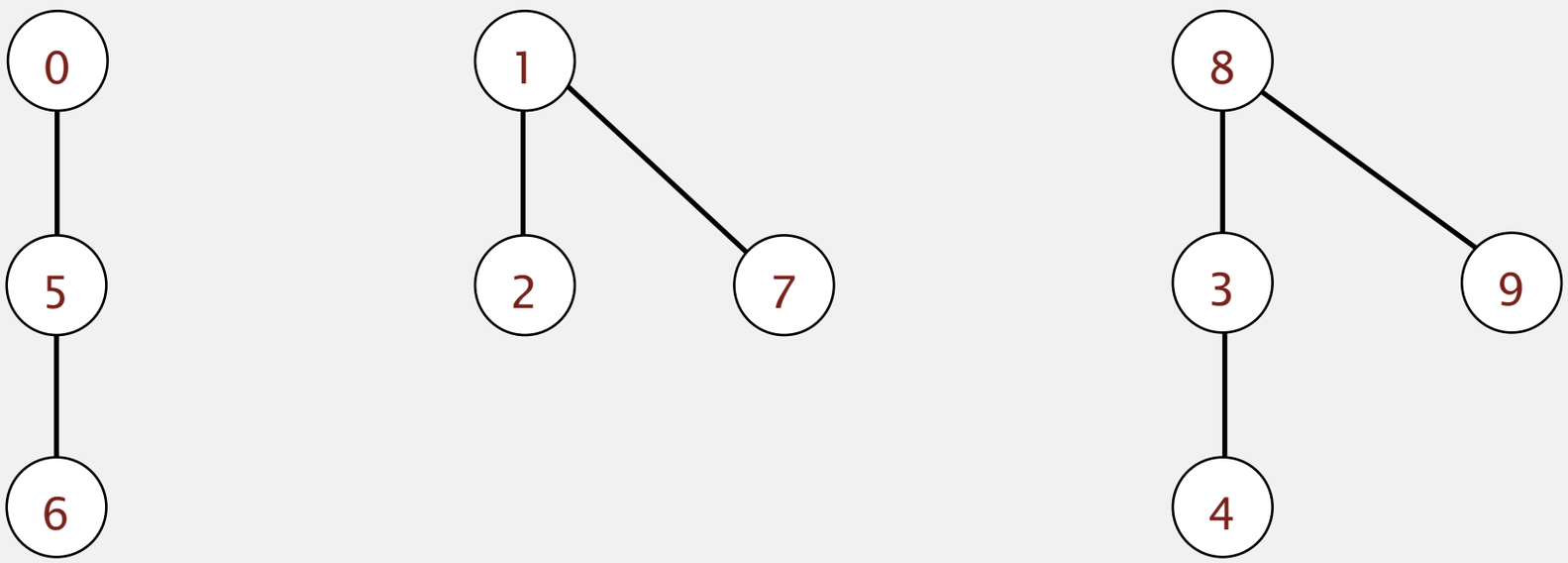


Quick-union demo



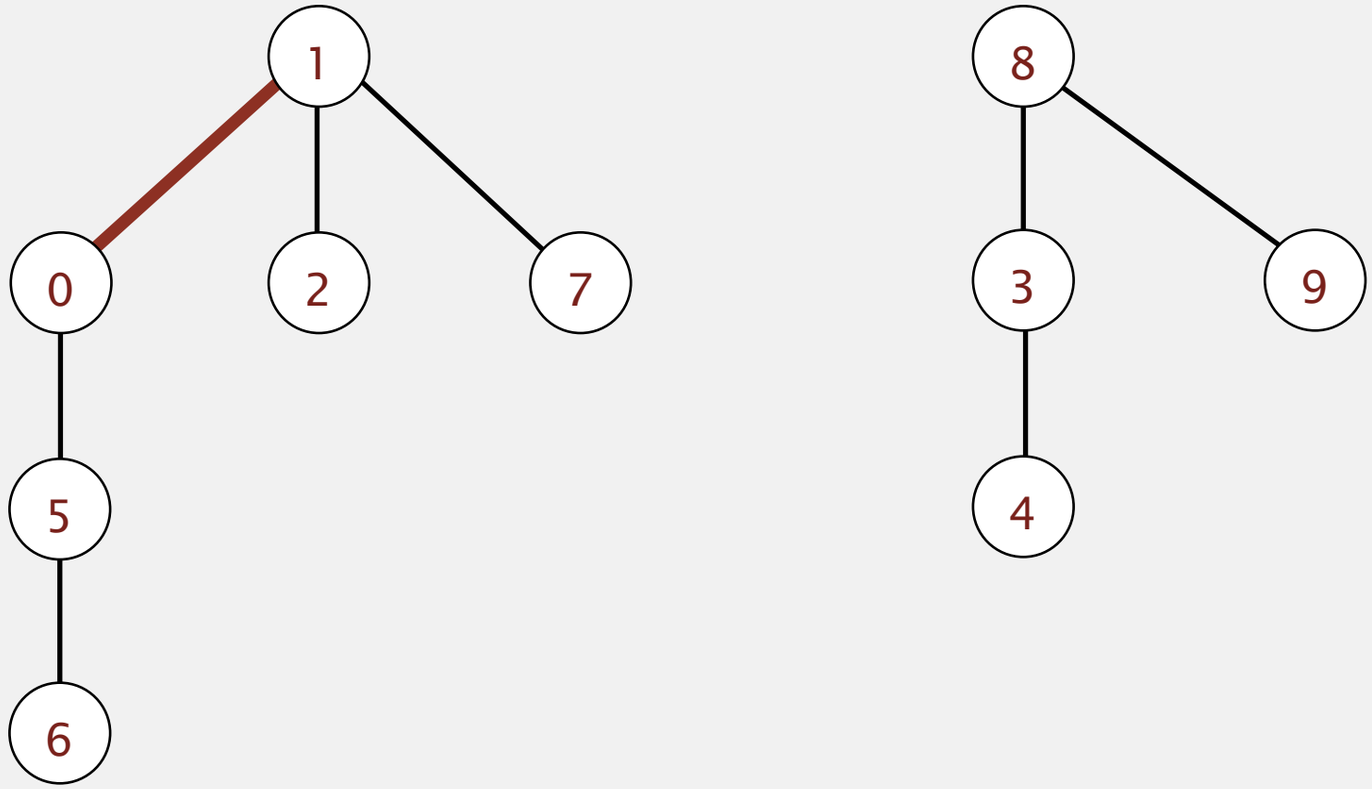
Quick-union demo

Union 6–1

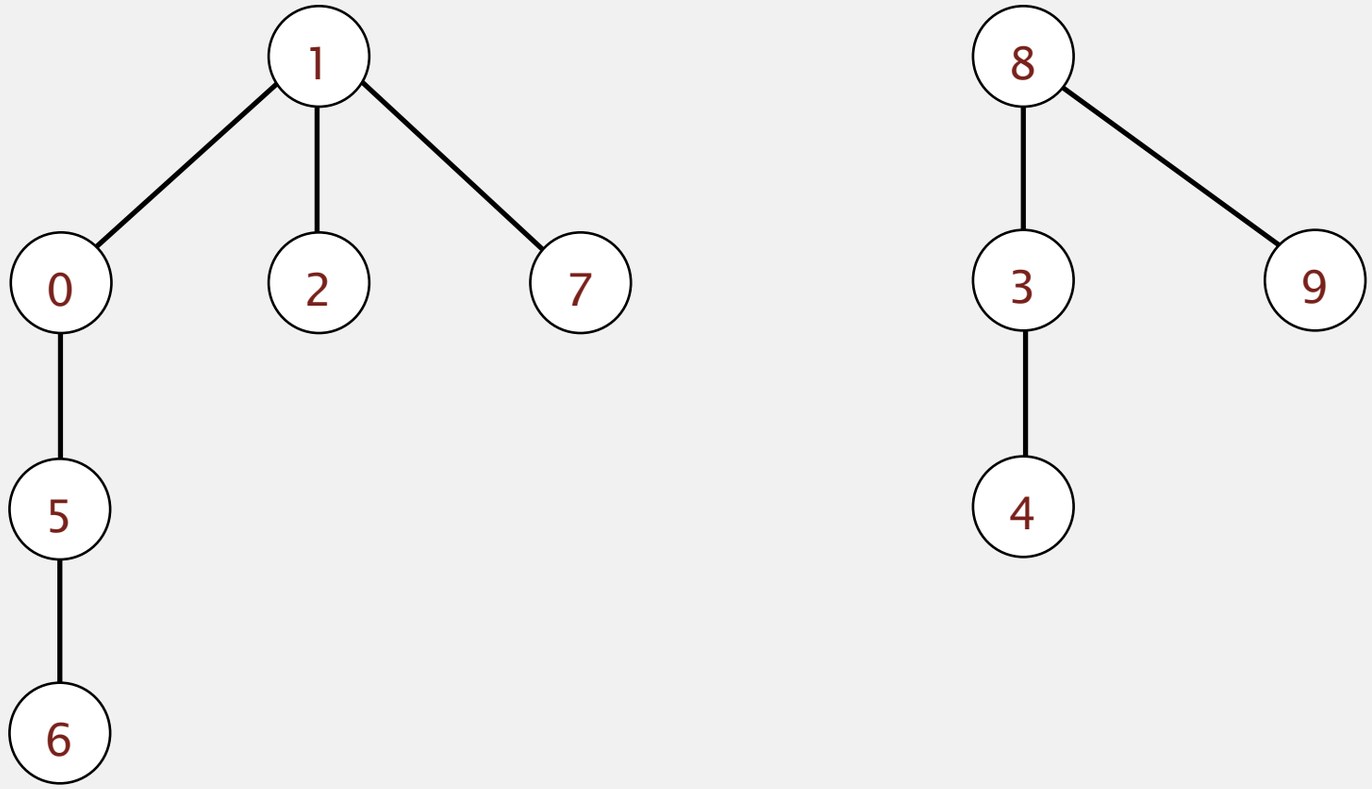


Quick-union demo

Union 6–1

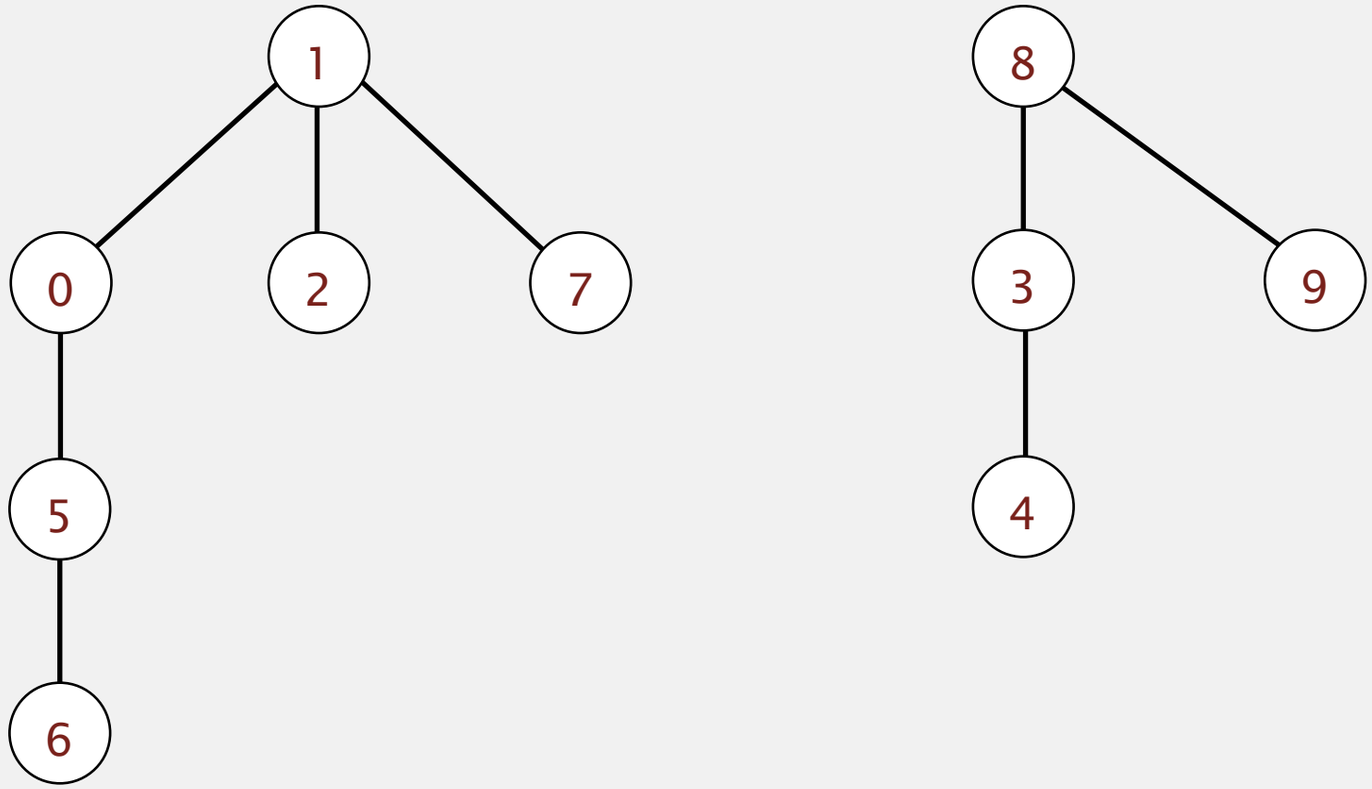


Quick-union demo



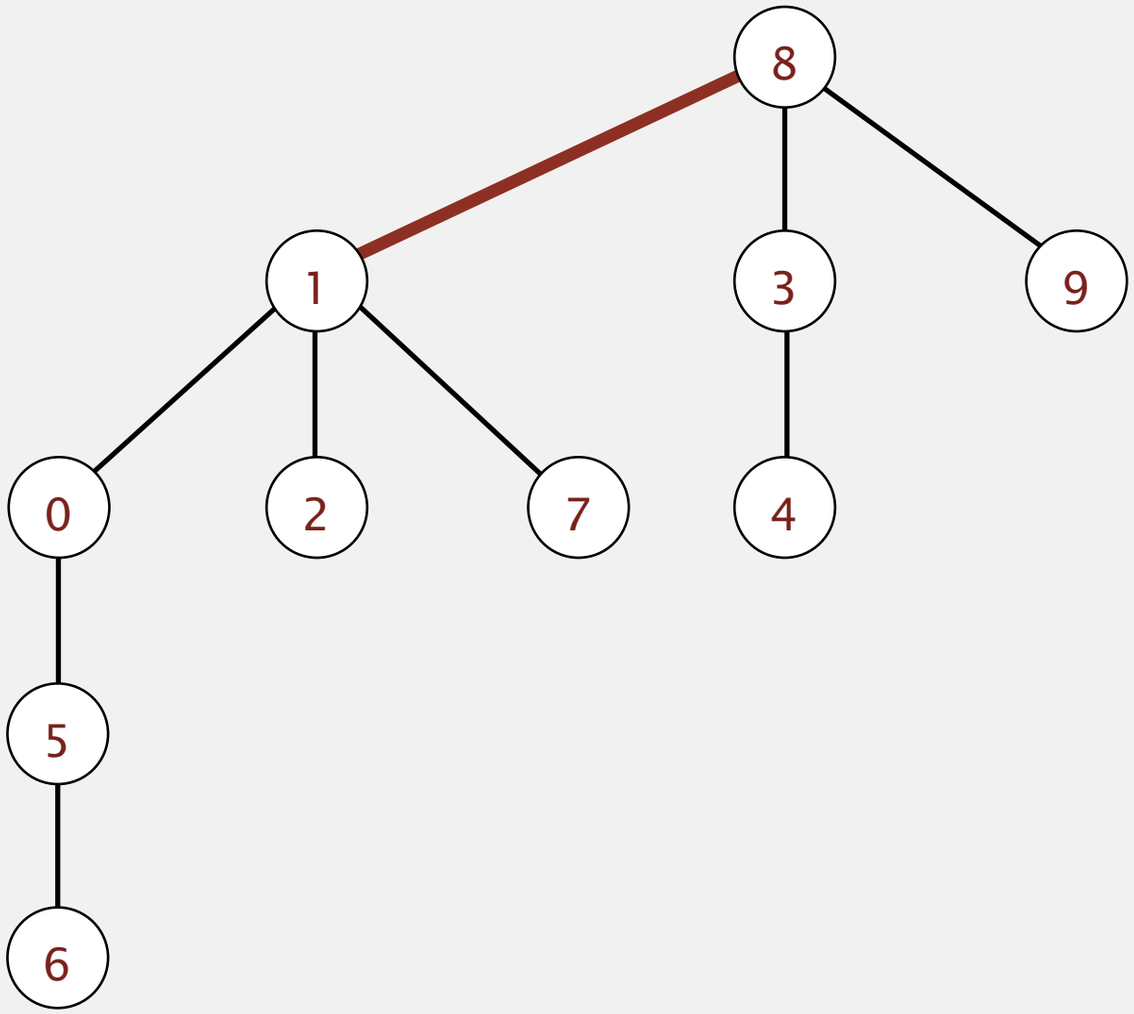
Quick-union demo

Union 5-3

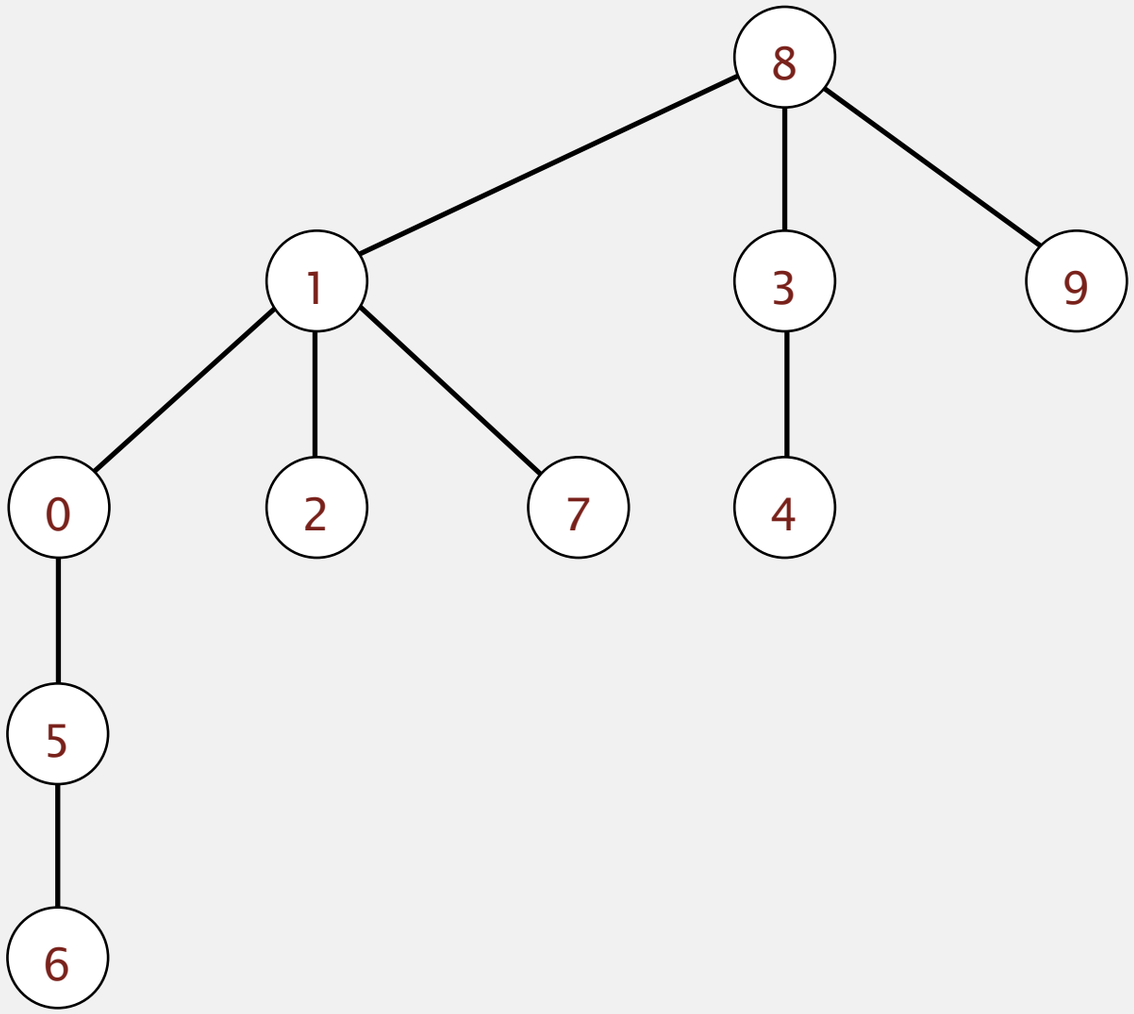


Quick-union demo

Union 5-3

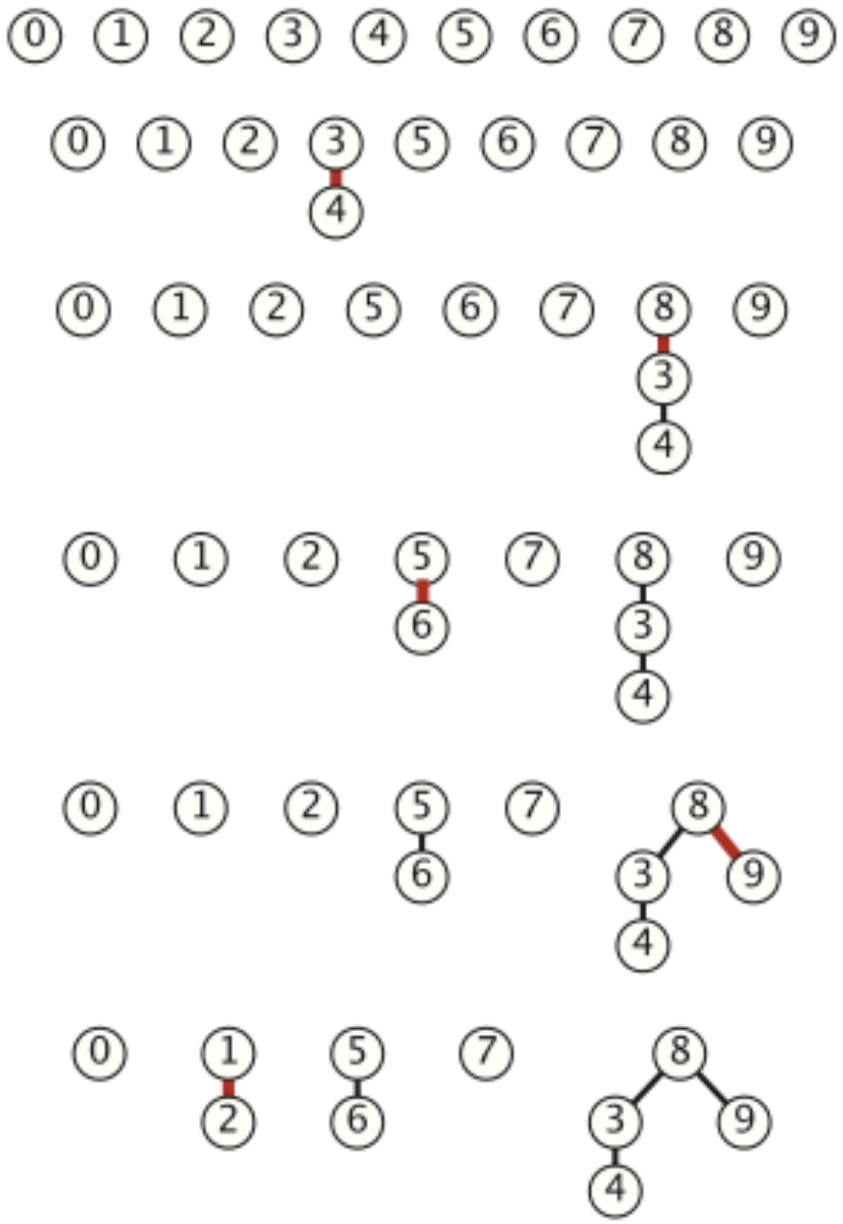


Quick-union demo



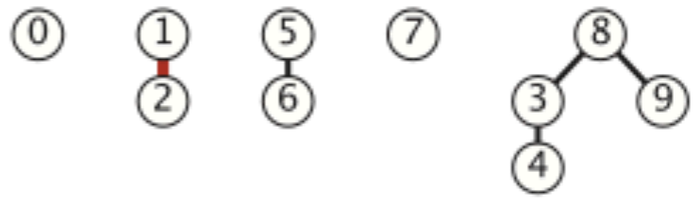
Quick-union example

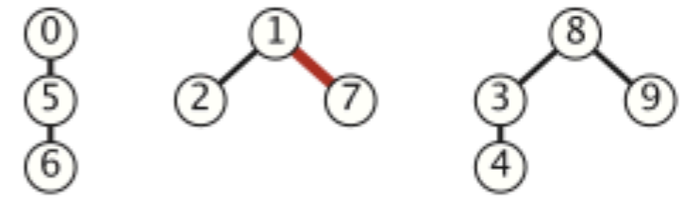
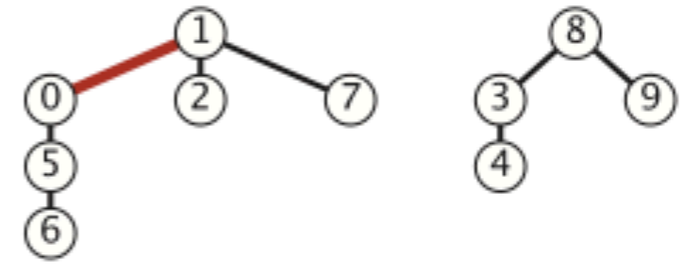
p	q	id[] 0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	3	5	6	7	8	9
6	5	0	1	2	8	3	5	6	7	8	9
		0	1	2	8	3	5	5	7	8	9
9	4	0	1	2	8	3	5	5	7	8	9
		0	1	2	8	3	5	5	7	8	8
2	1	0	1	2	8	3	5	5	7	8	8
		0	1	1	8	3	5	5	7	8	8



Quick-union example

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



Quick-union: Java implementation

```
public class QuickUnionUF
{
   private int[] id;

   public QuickUnionUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   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.

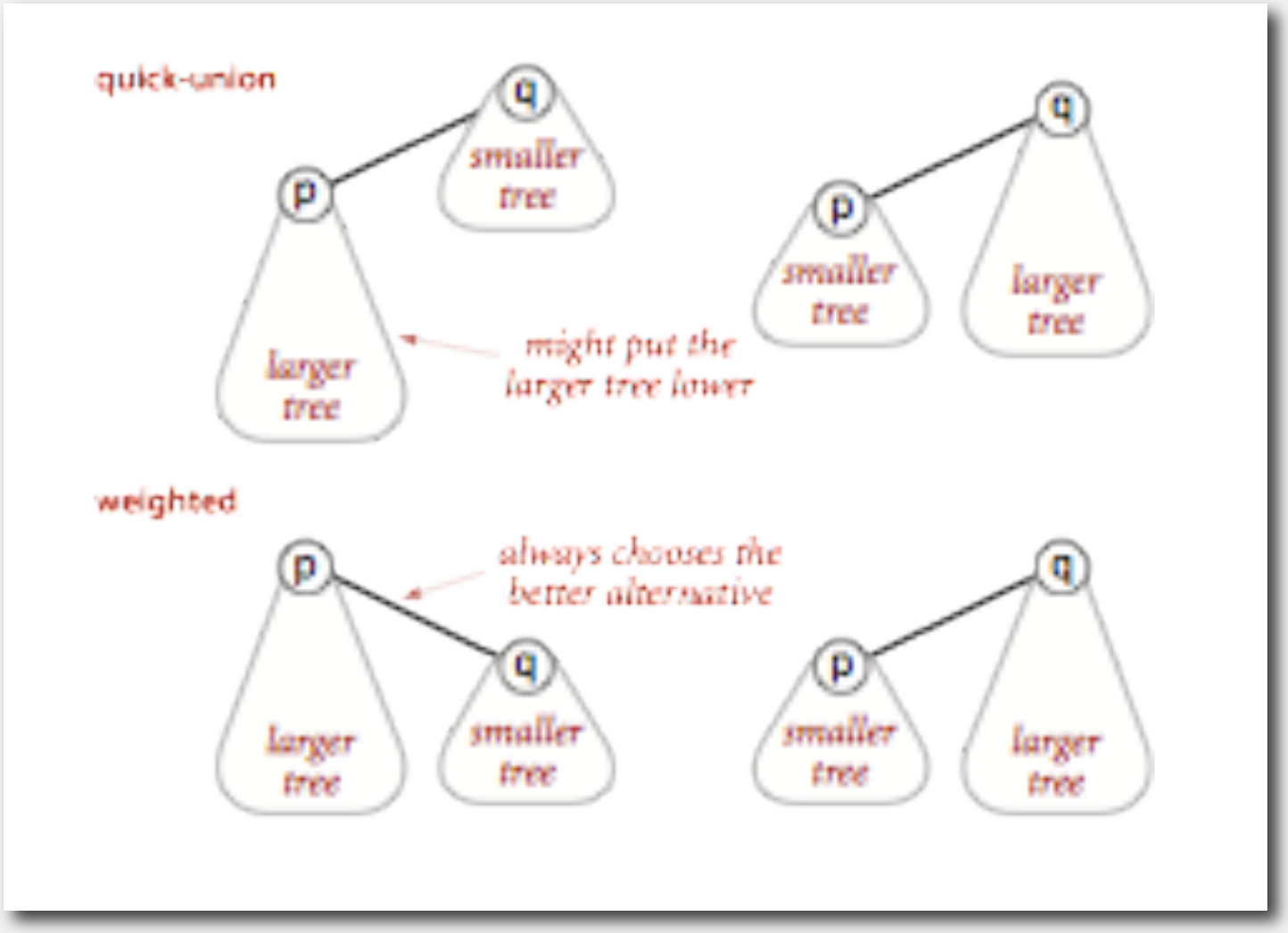
- Trees can get tall.
- Find too expensive (could be N array accesses).

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

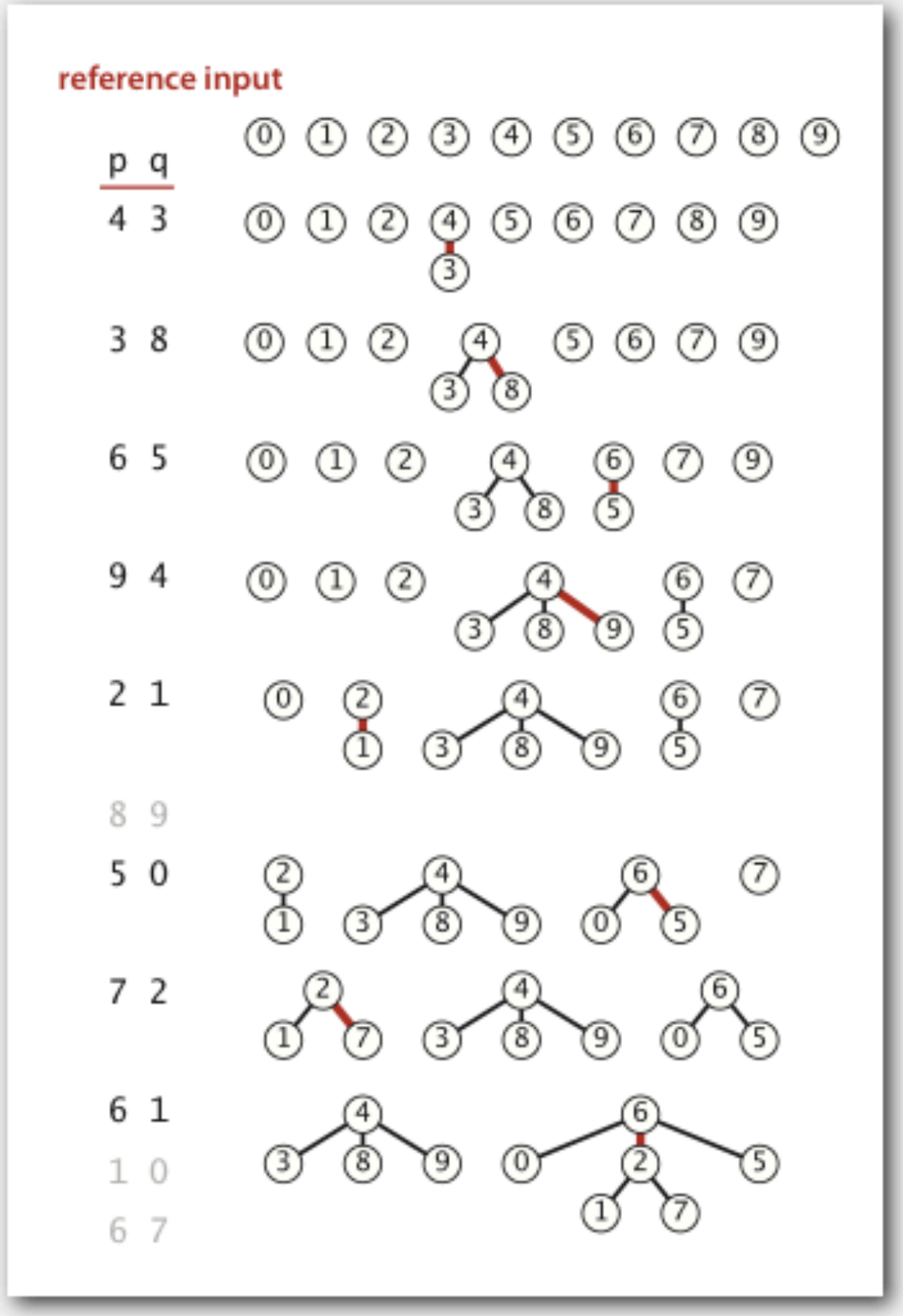
Improvement 1: weighting

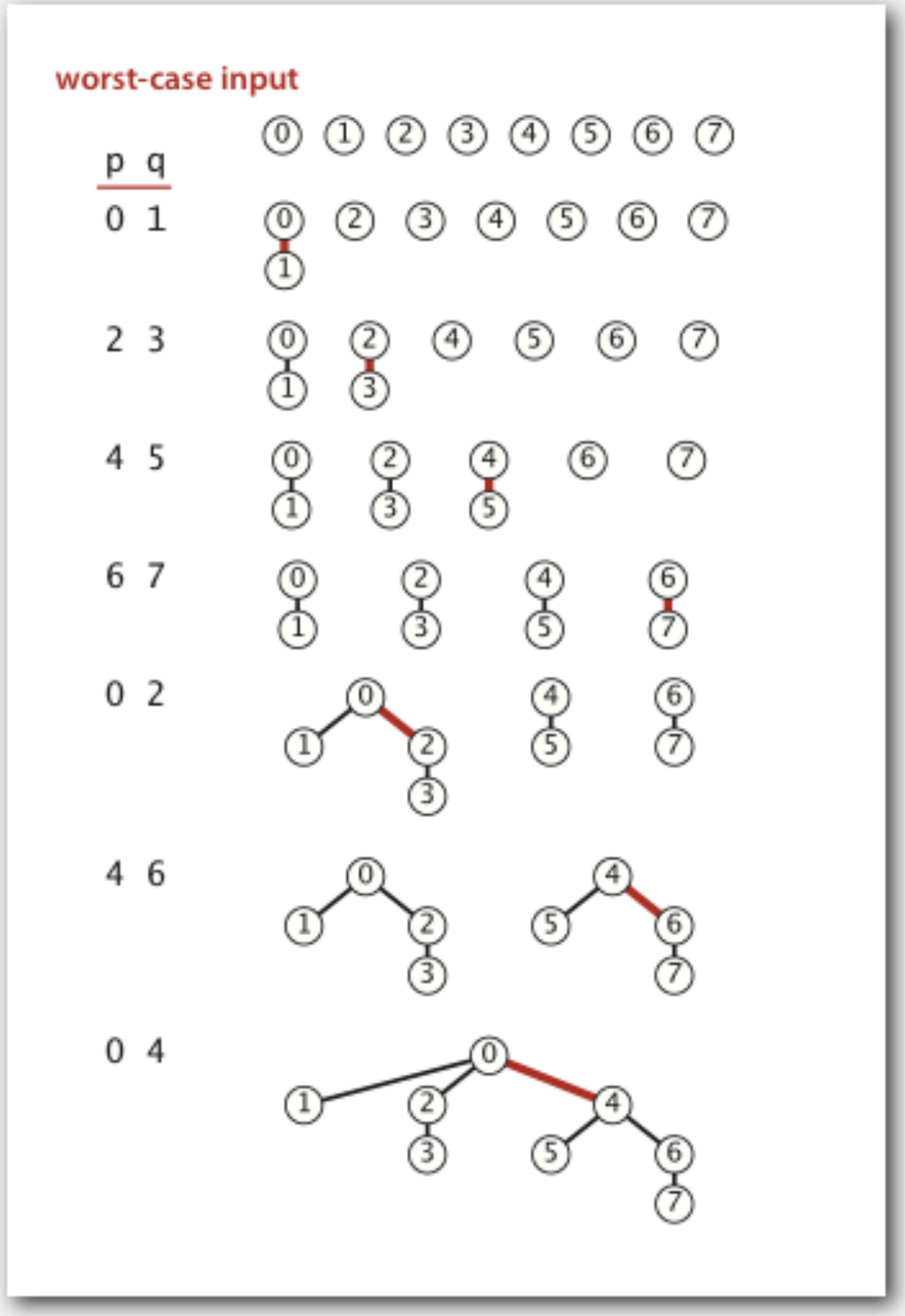
Weighted quick-union.

- 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.

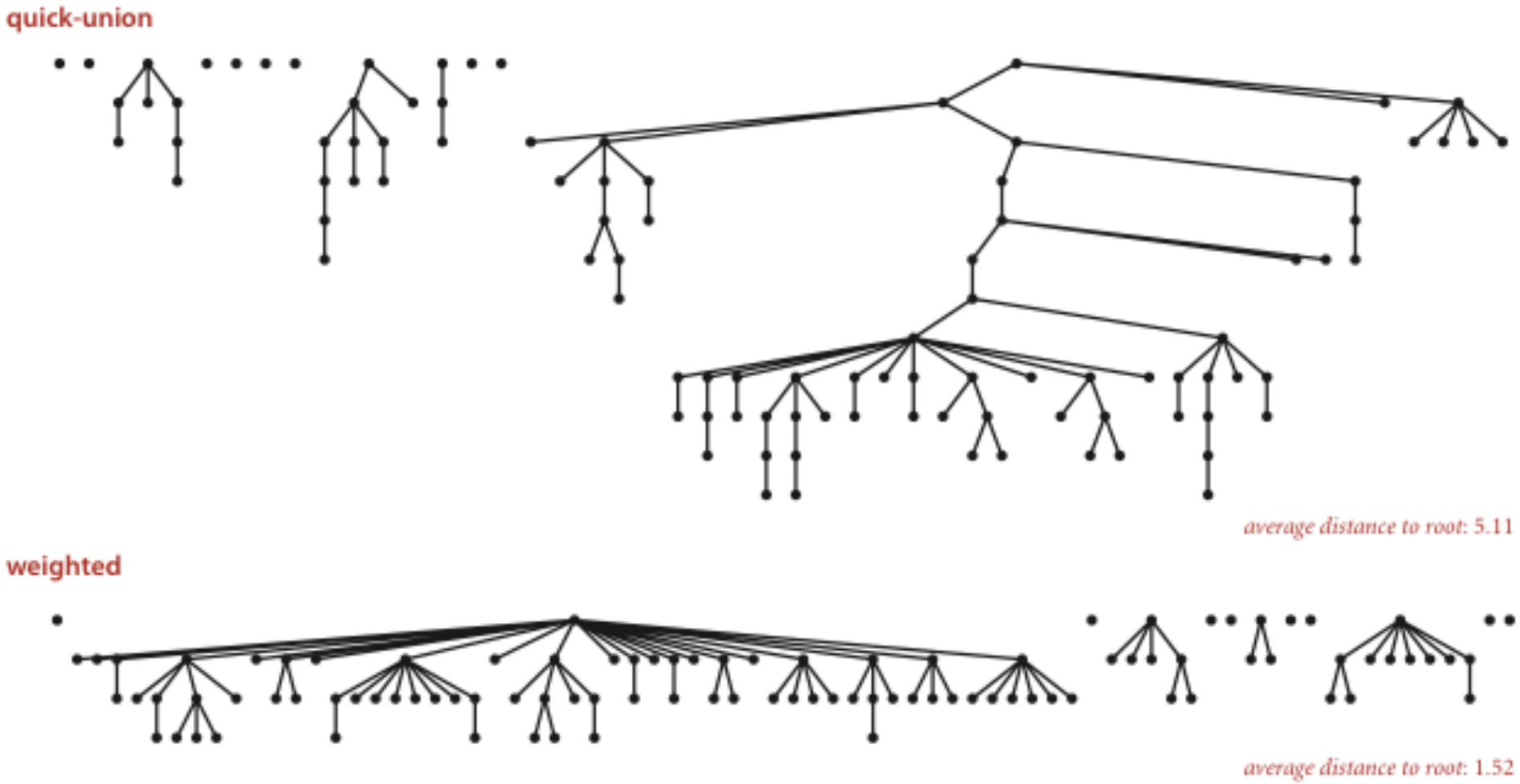


Weighted quick-union examples





Quick-union and weighted quick-union example



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.

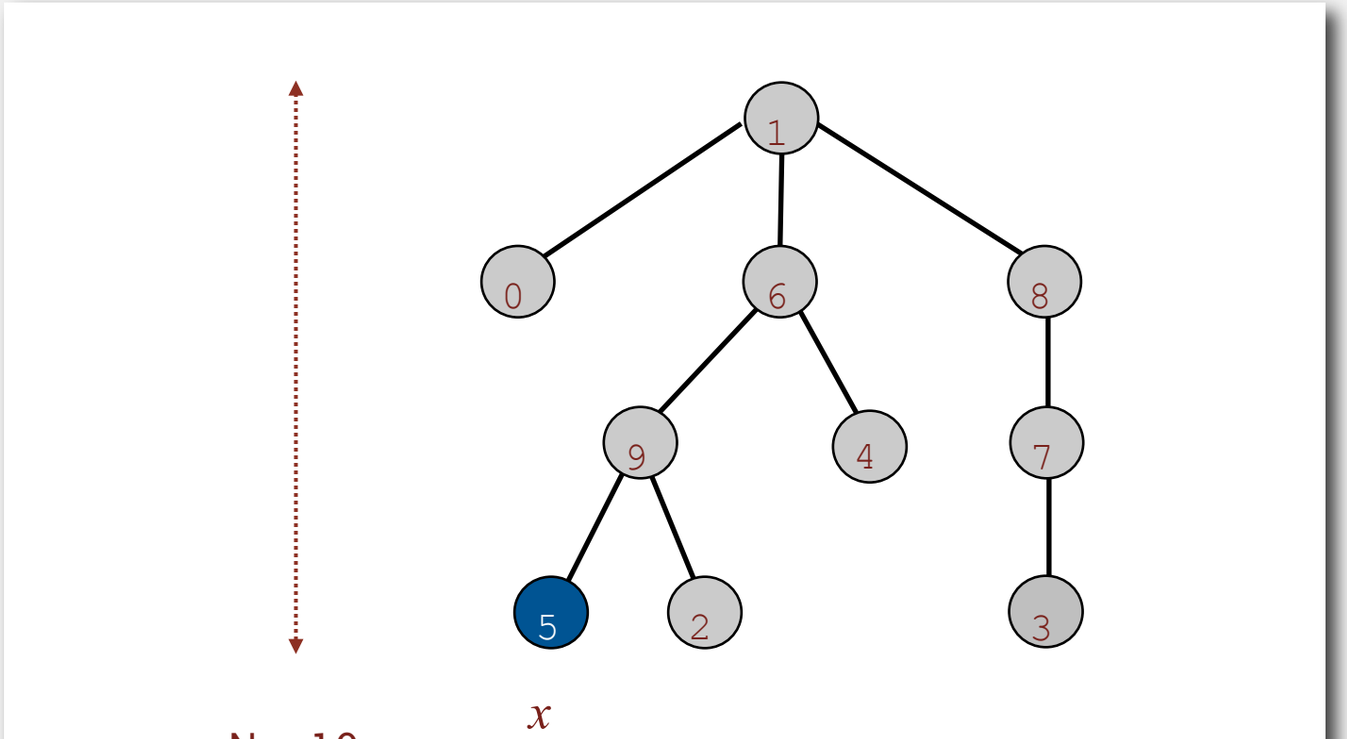
```
int i = root(p);
int j = root(q);
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

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 $\lg N$.



Weighted quick-union analysis

Running time.

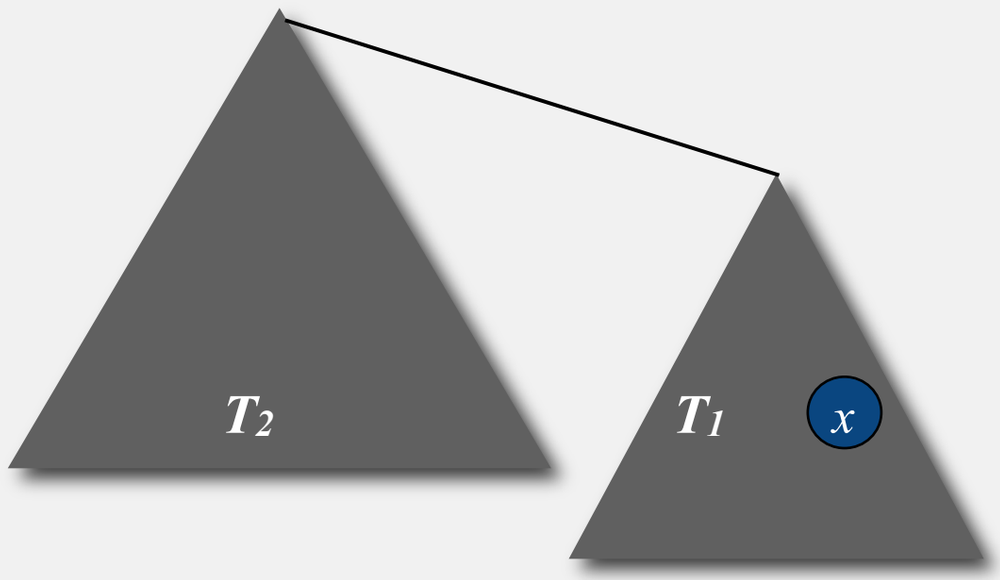
- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

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

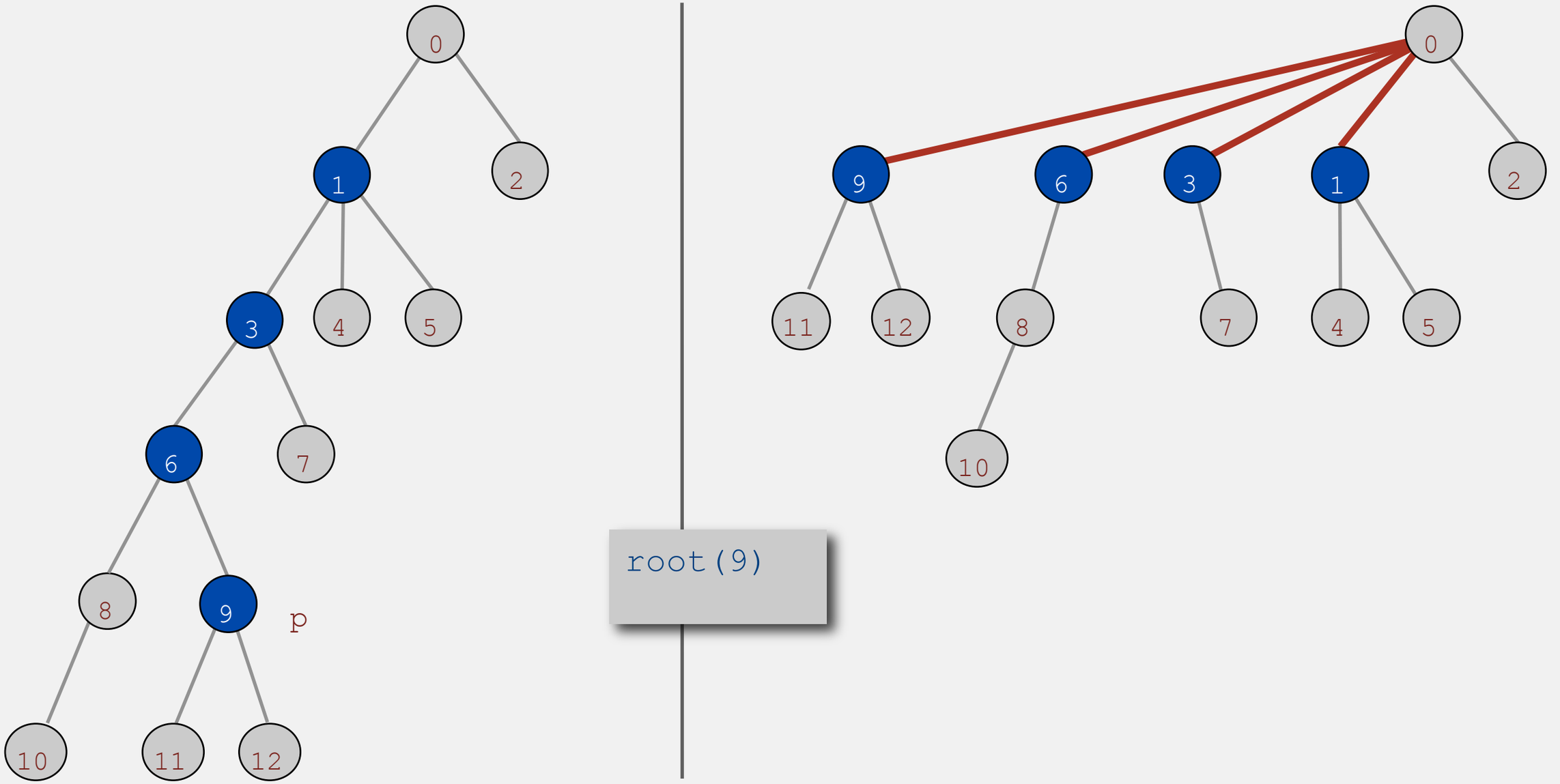
† includes cost of finding roots

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.



Path compression: Java implementation

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

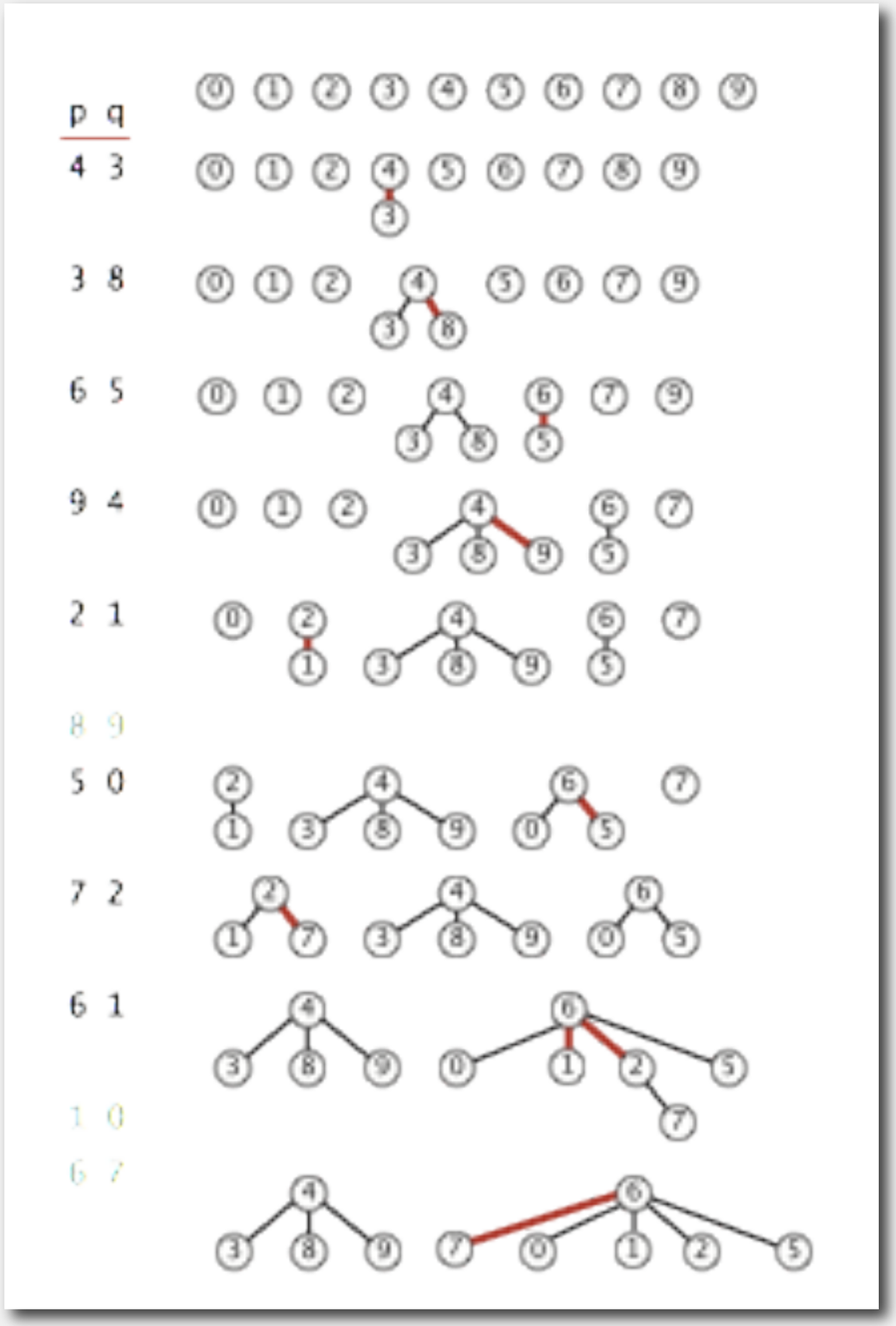
Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length).

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

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression 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!
- Analysis can be improved to $N + M \alpha(M, N)$. see COS 423

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.

Amazing fact. No linear-time algorithm exists.

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

lg* function

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

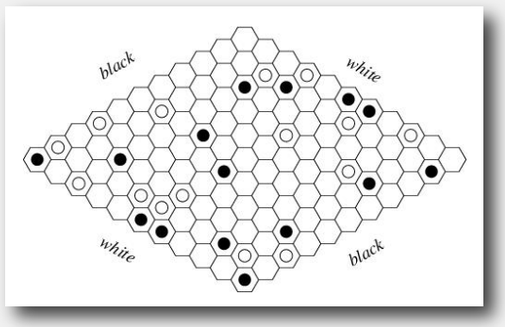
Ex. [10^9 unions and finds with 10^9 objects]

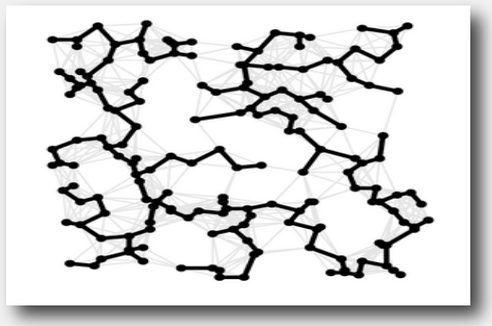
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

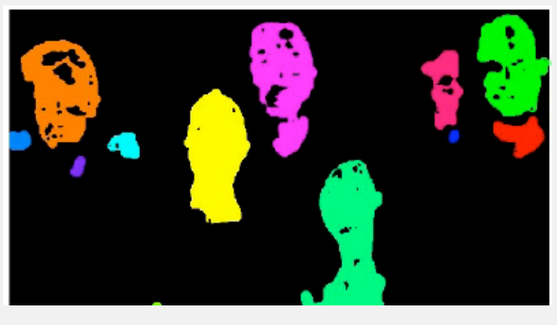
- dynamic connectivity
- quick find
- quick union
- improvements
- **applications**

Union-find applications

- Percolation. ← see also Assignment 1
- 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.



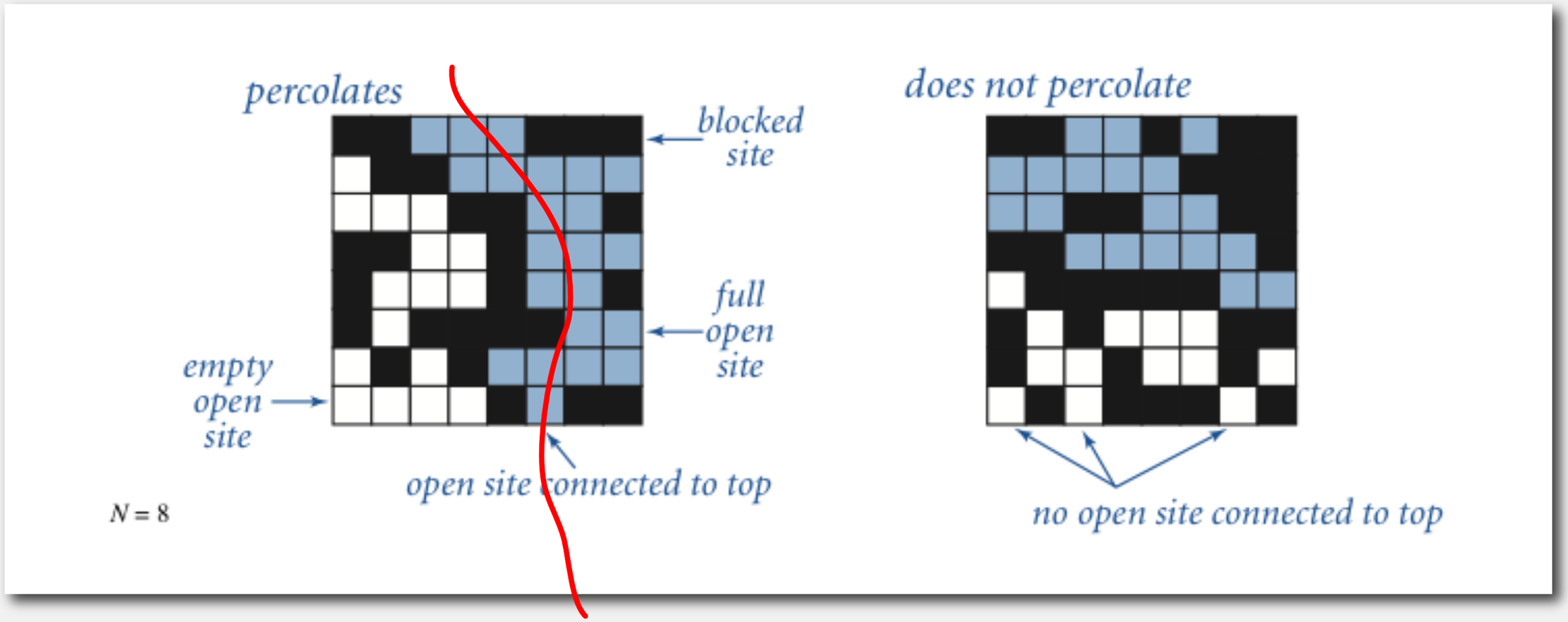




Percolation

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

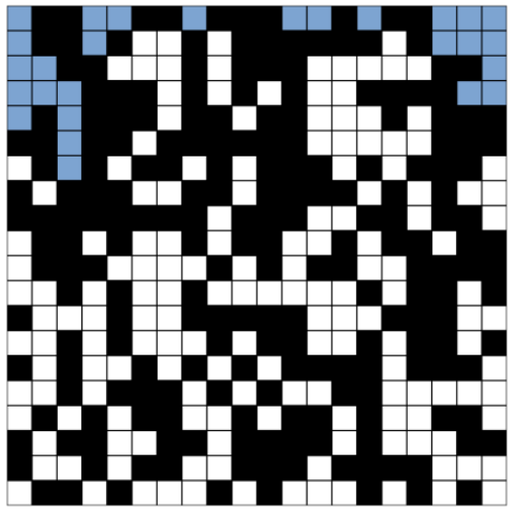
A model for many physical systems:

- 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

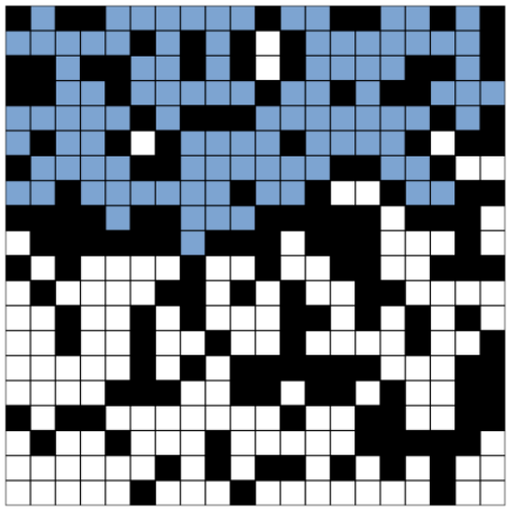
Likelihood of percolation

Depends on site vacancy probability p.



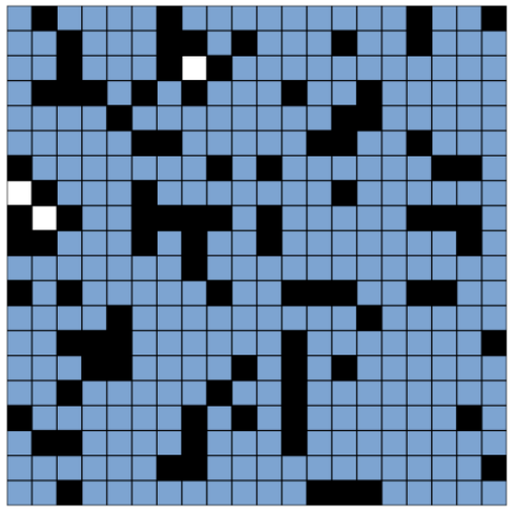
p low (0.4)

does not percolate



p medium (0.6)

percolates?



p high (0.8)

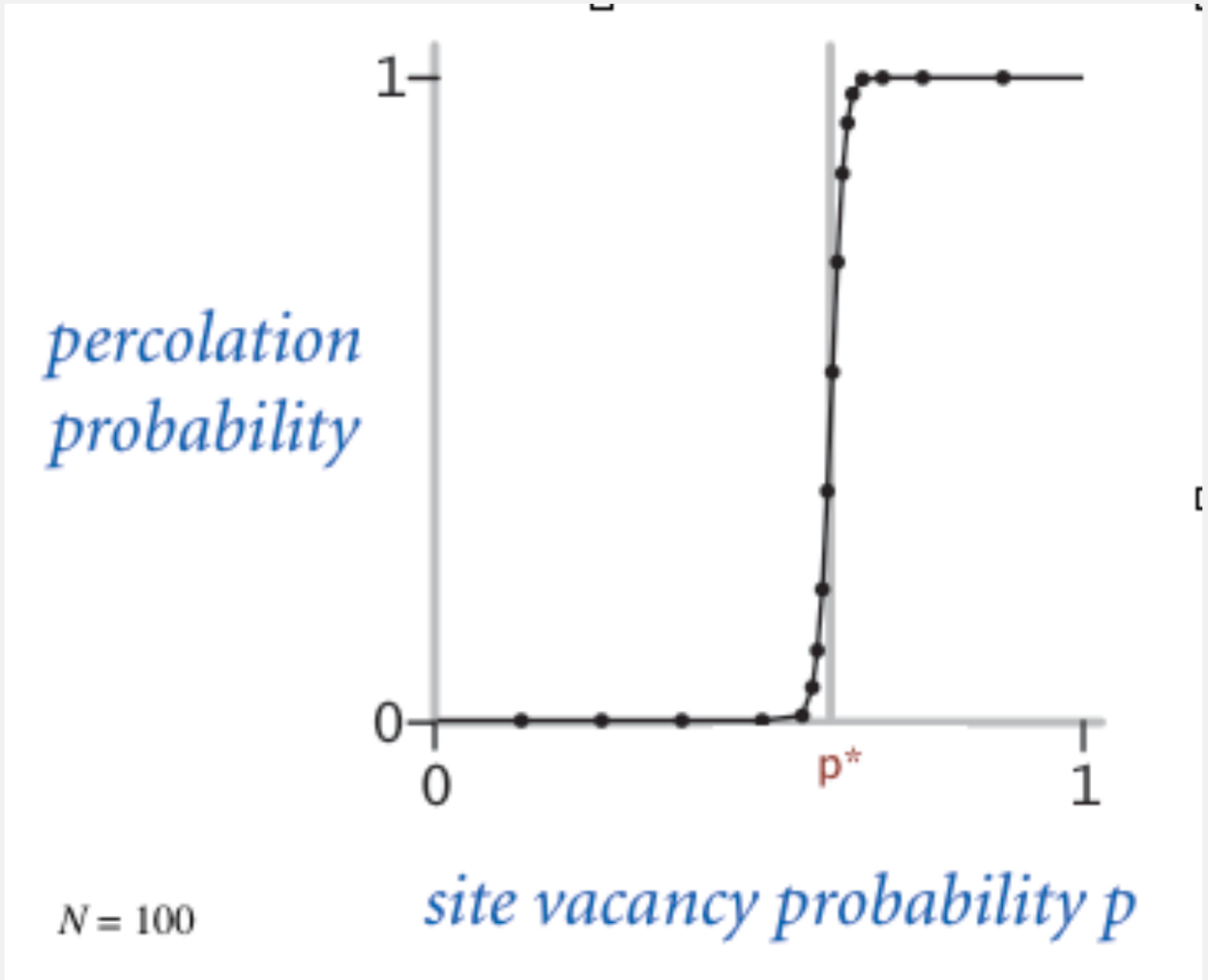
percolates

Percolation phase transition

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

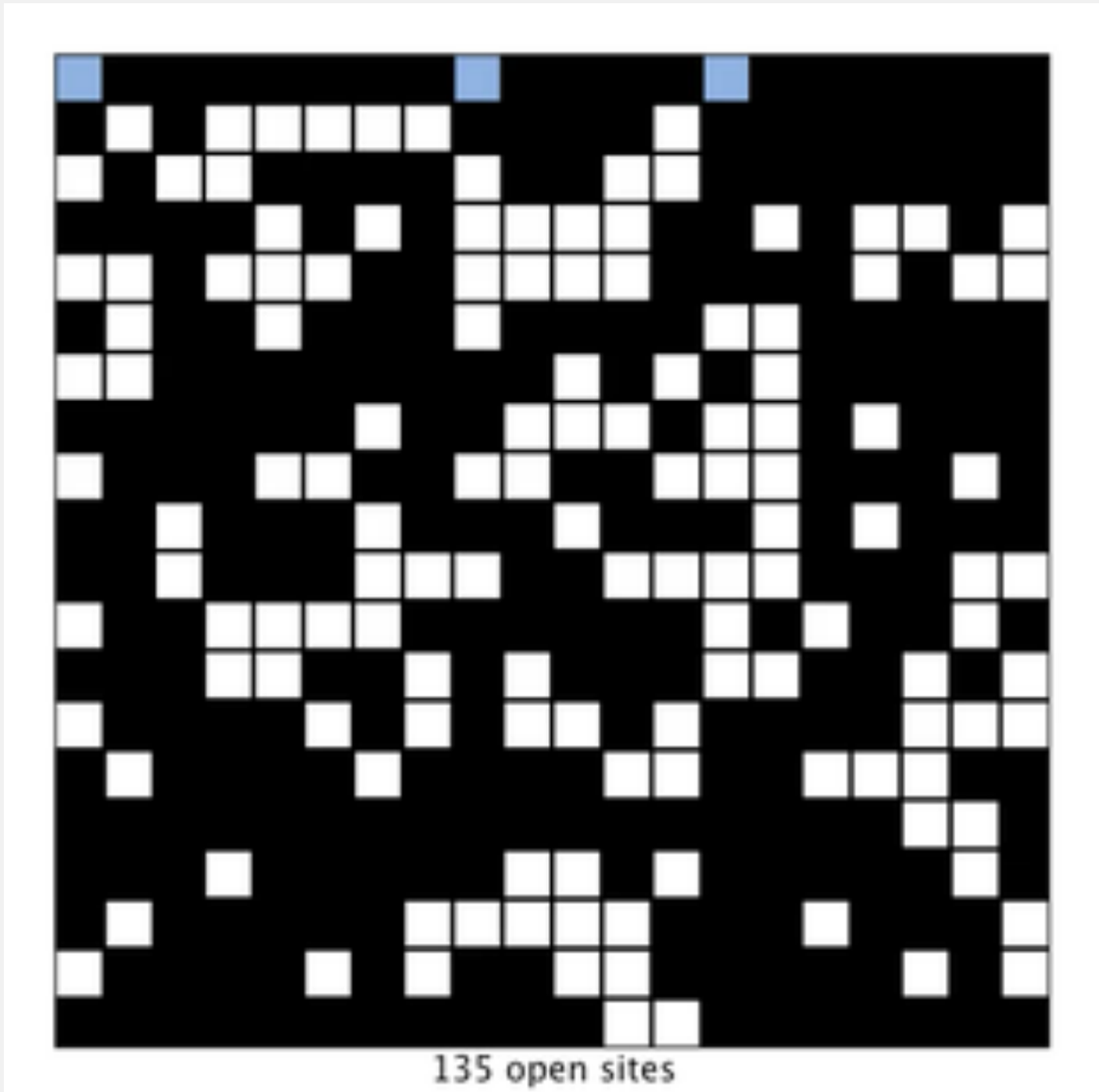
- $p > p^*$: almost certainly percolates.
- $p < p^*$: almost certainly does not percolate.

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^*.



$N = 20$

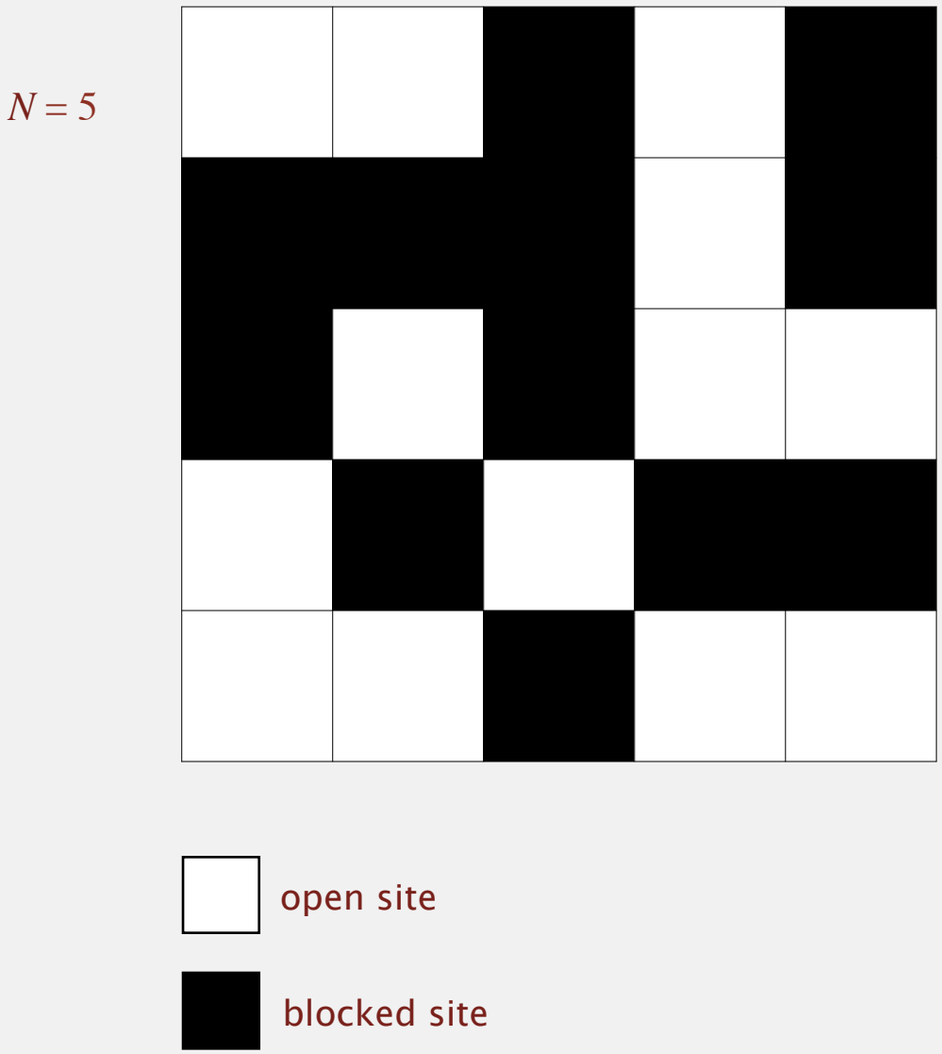
full open site
(connected to top)

empty open site
(not connected to top)

blocked site

Dynamic connectivity solution to estimate percolation threshold

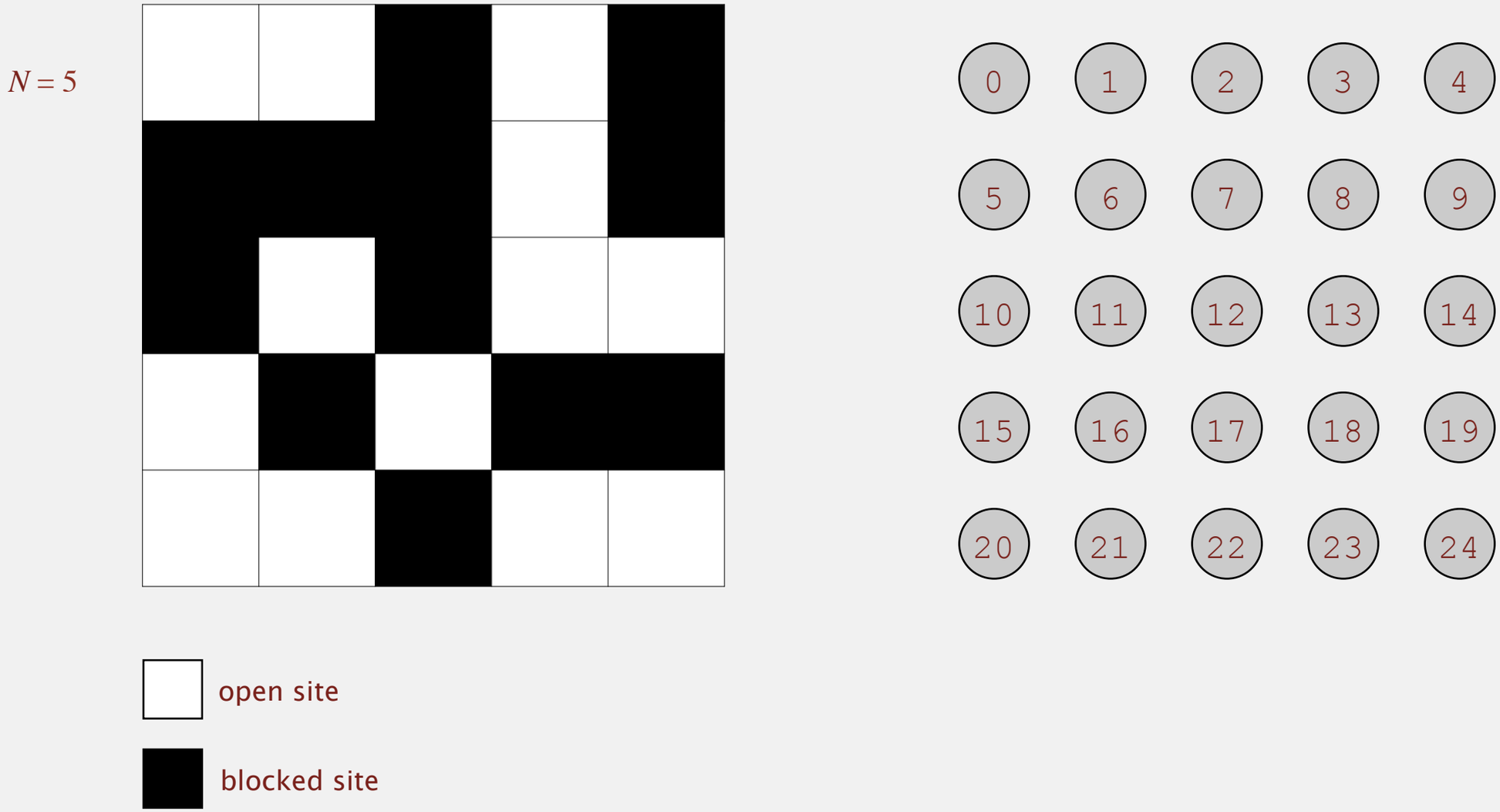
Q. How to check whether an N-by-N system percolates?



Dynamic connectivity solution to estimate percolation threshold

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

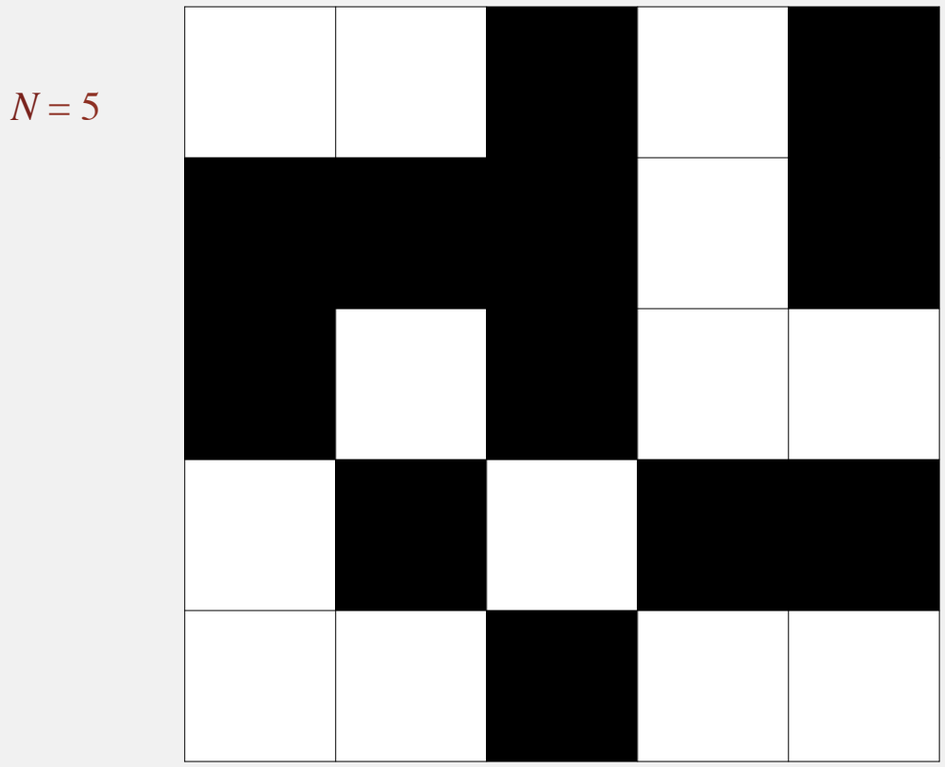
- Create an object for each site and name them 0 to $N^2 - 1$.

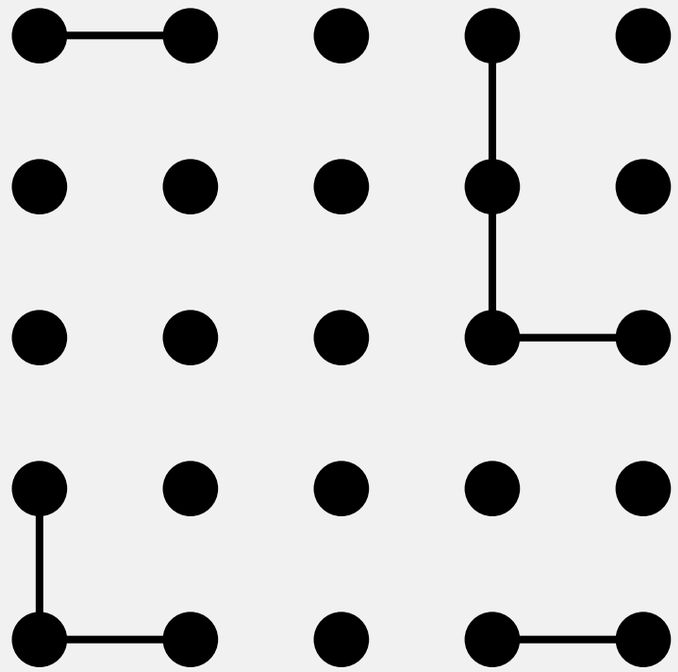


Dynamic connectivity solution to estimate percolation threshold

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.



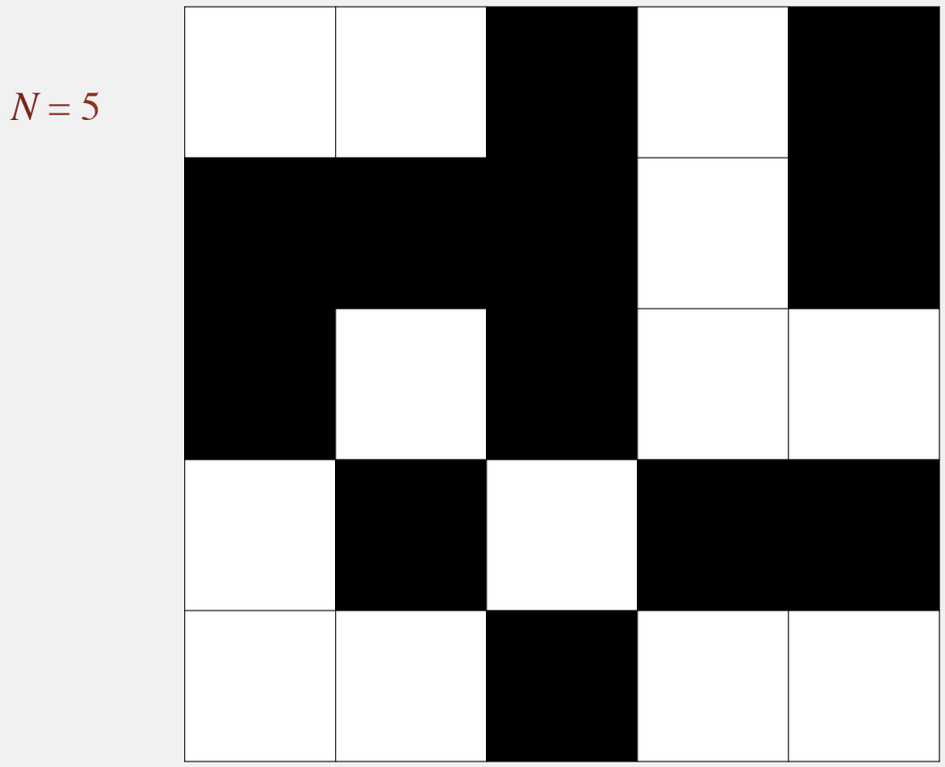


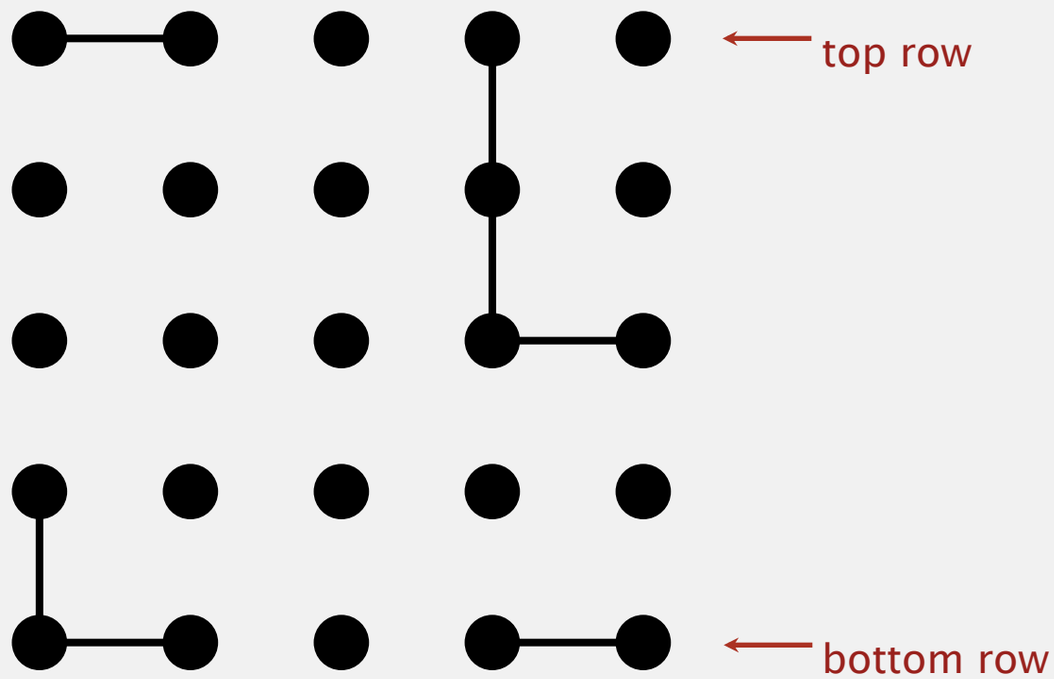
Dynamic connectivity solution to estimate percolation threshold

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()



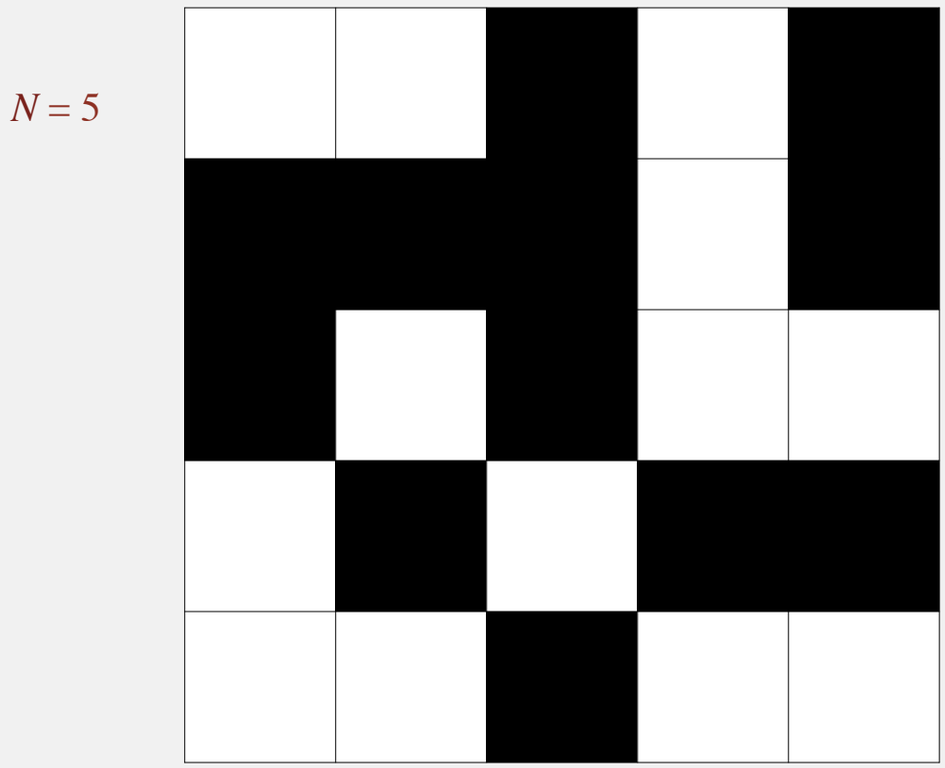


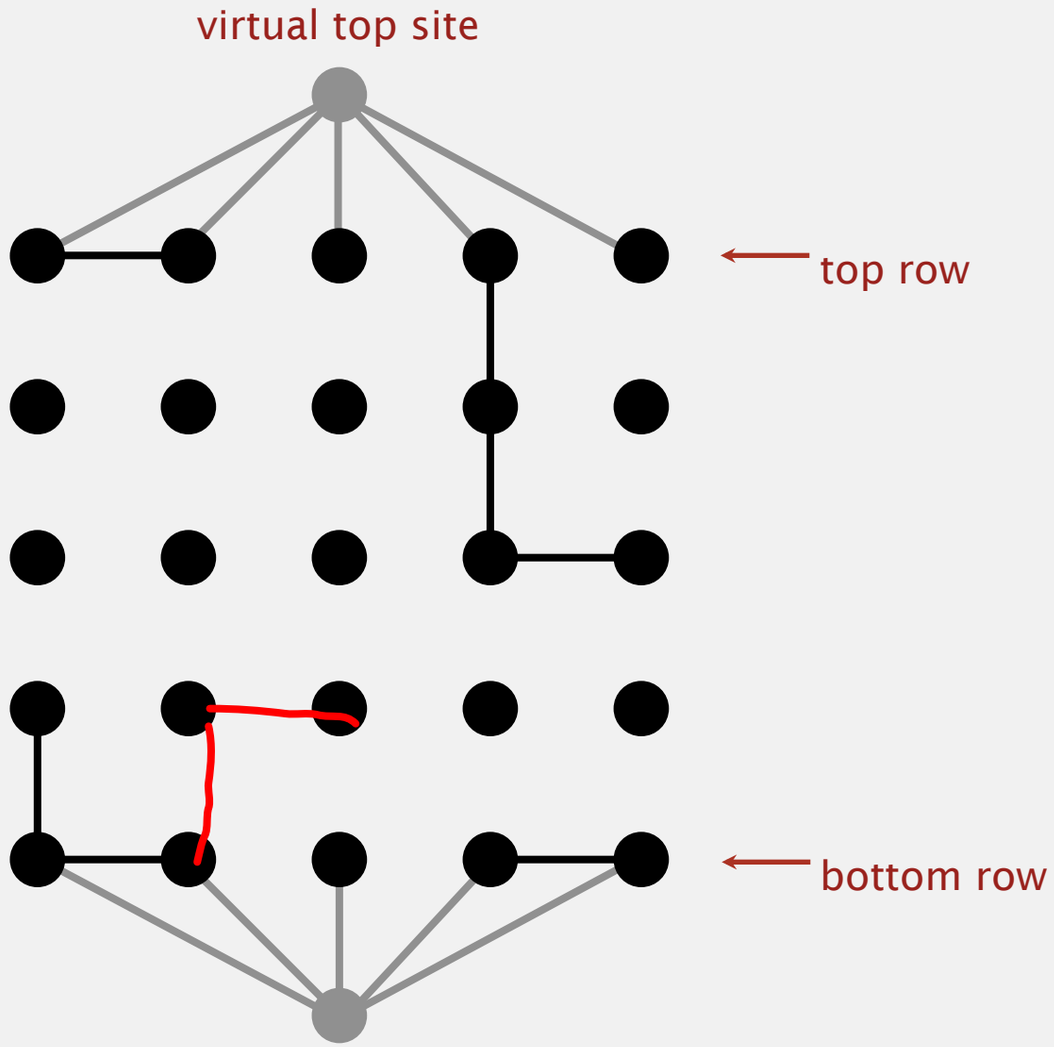
Dynamic connectivity solution to estimate percolation threshold

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()





open site

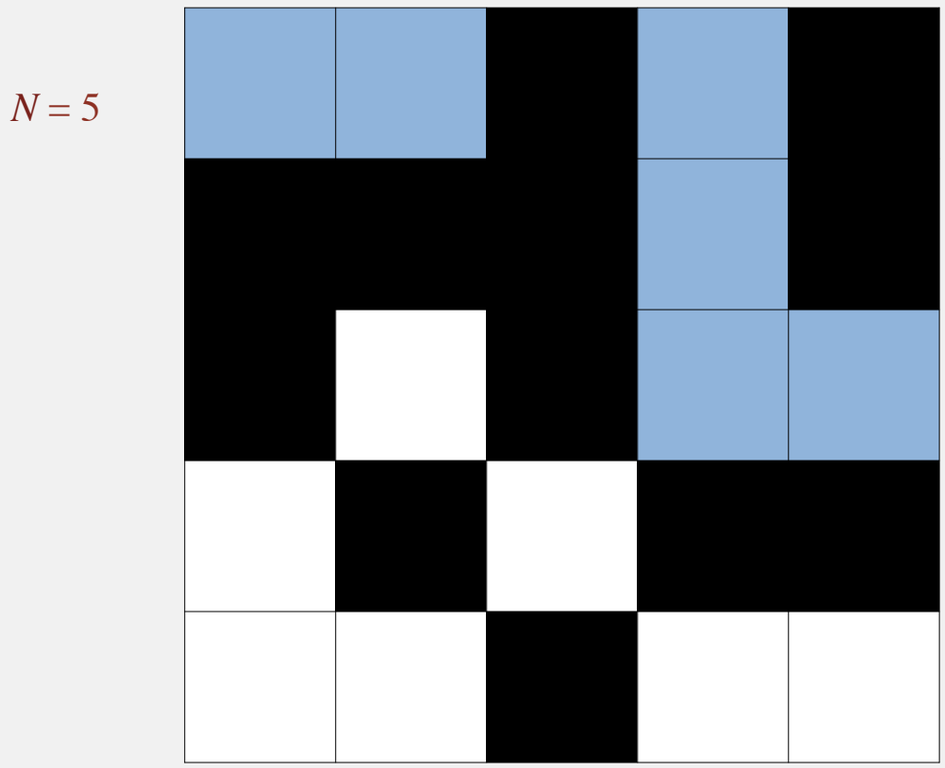
blocked site

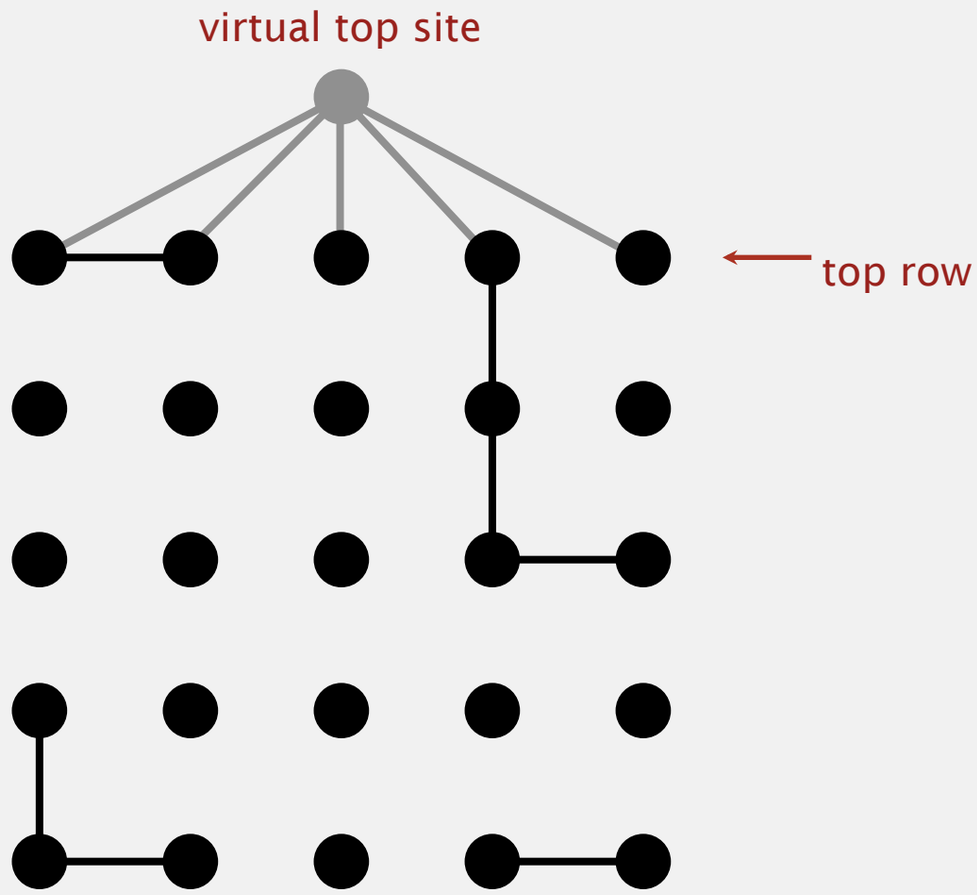
Dynamic connectivity solution to estimate percolation threshold

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

- Percolates iff virtual top site is connected to virtual bottom site.
- Open site is full iff connected to virtual top site (don't use virtual bottom).

needed only for visualization





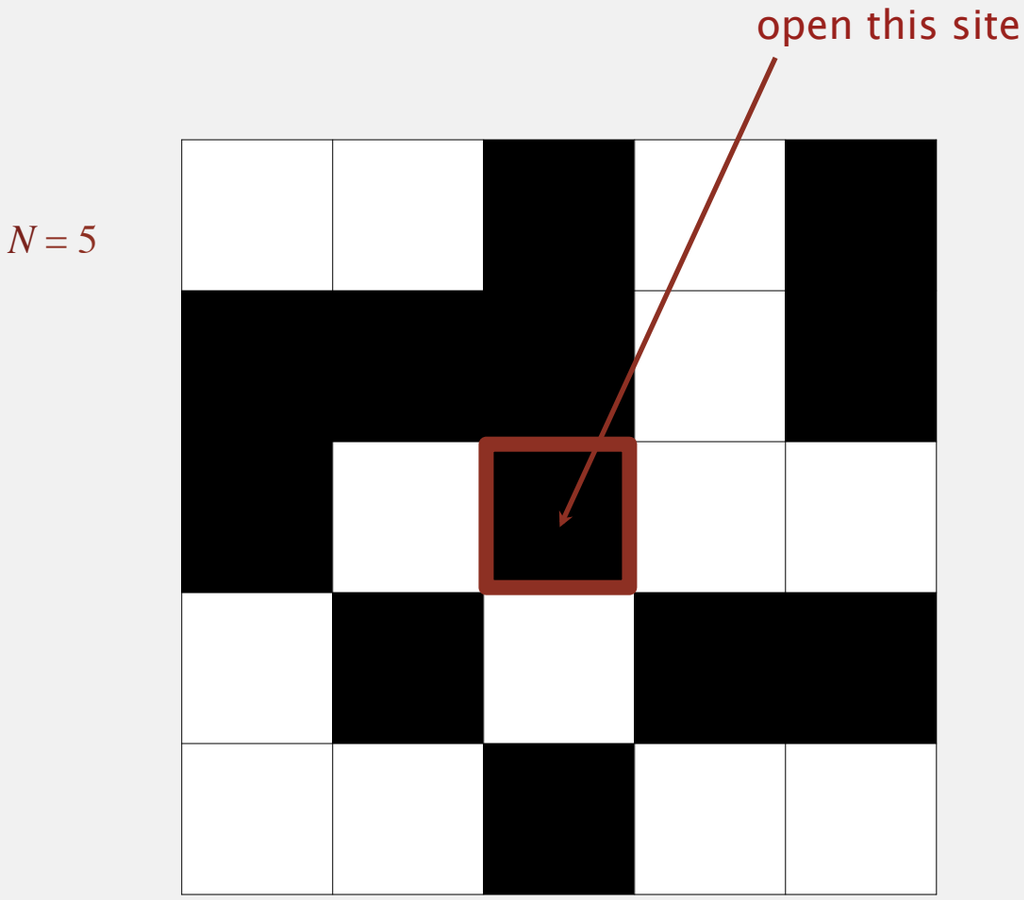
empty open site
(not connected to top)

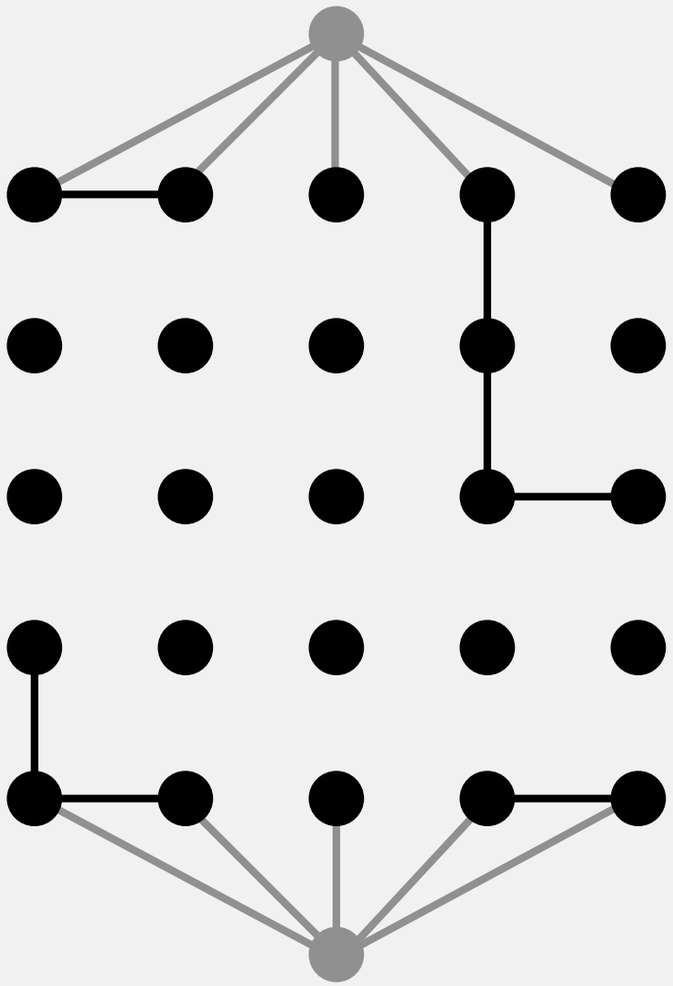
full open site
(connected to top)

blocked site

Dynamic connectivity solution to estimate percolation threshold

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






open site

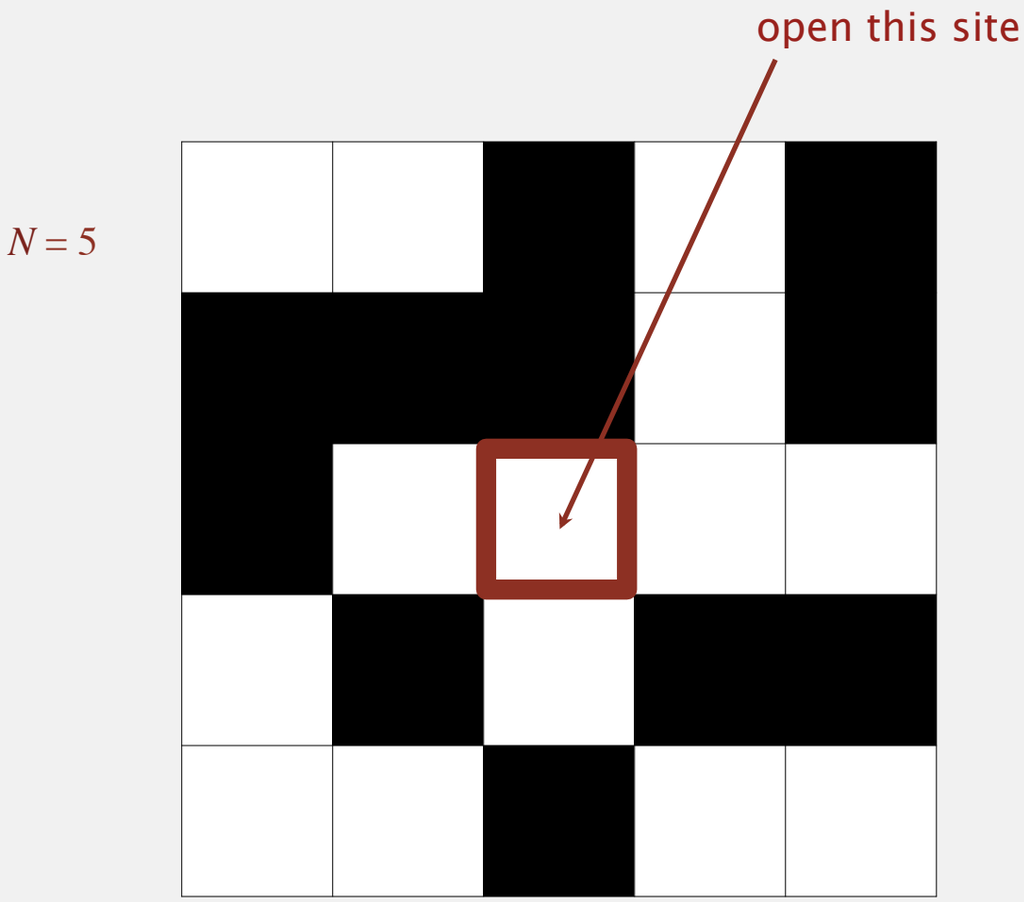

blocked site

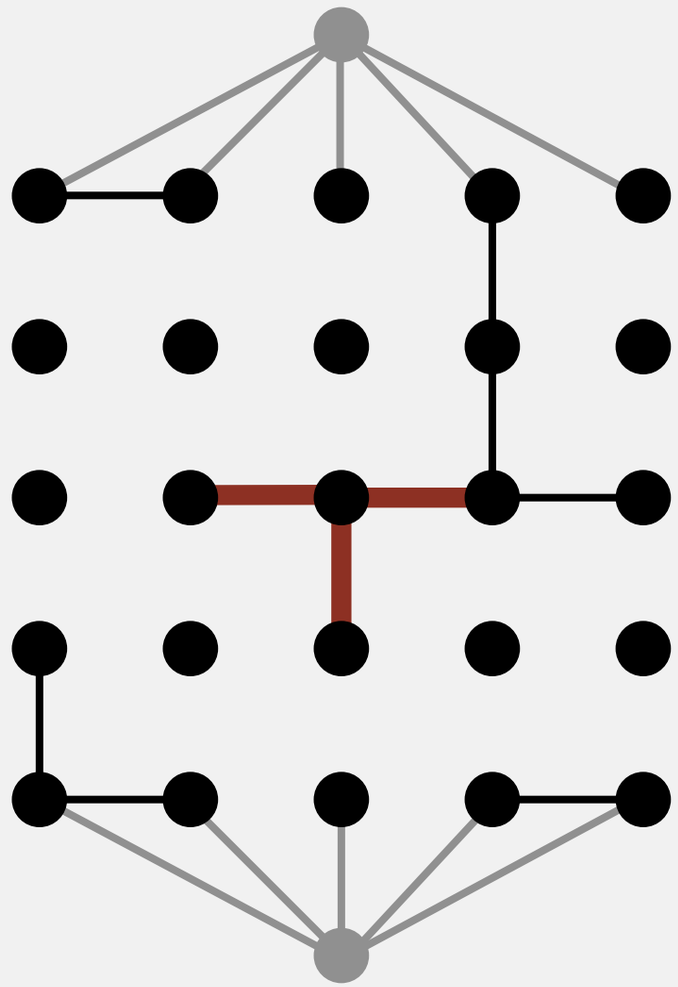
Dynamic connectivity solution to estimate percolation threshold

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 site

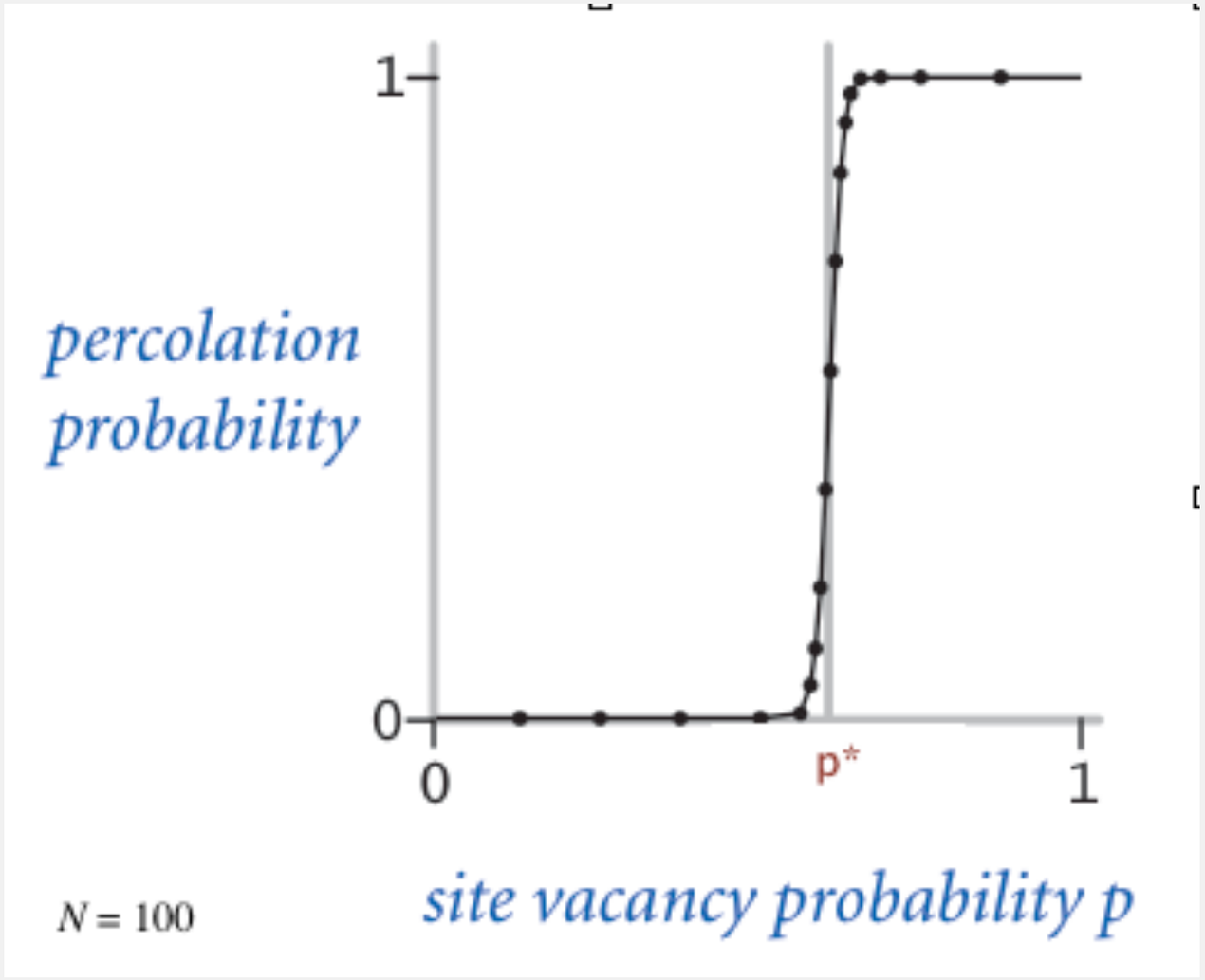
blocked site

Percolation threshold

Q. What is percolation threshold p^* ?

A. About 0.592746 for large square lattices.

constant know only via simulation

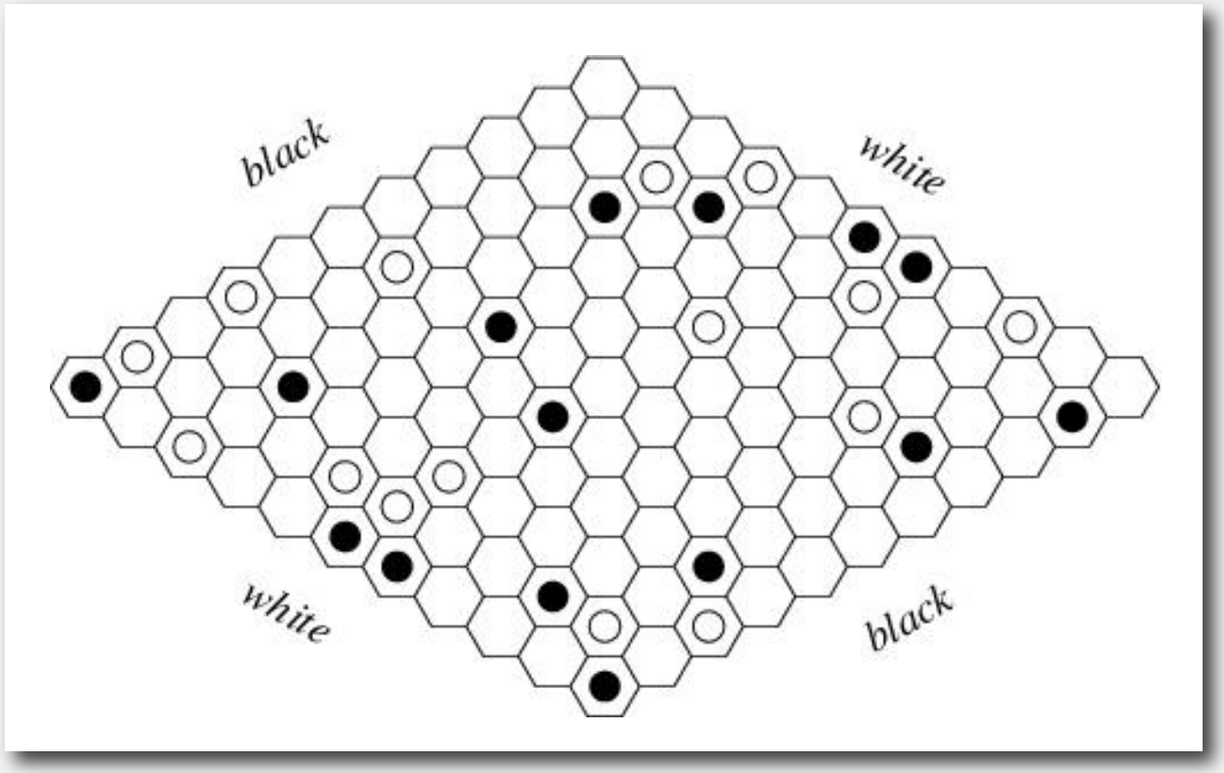


Fast algorithm enables accurate answer to scientific question.

Hex

Hex. [Piet Hein 1942, John Nash 1948, Parker Brothers 1962]

- Two players alternate in picking a cell in a hexagonal grid.
- Black: make a black path from upper left to lower right.
- White: make a white path from lower left to upper right.



http://mathworld.wolfram.com/GameofHex.html

Union-find application. Algorithm to detect when a player has won.

Subtext of 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.