Putting the Science Back into Computer Science

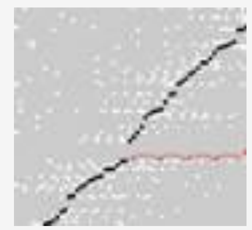
Robert Sedgewick
Princeton University

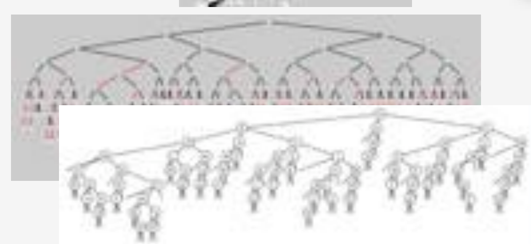
The scientific method

is essential in applications of computation

A personal opinion formed on the basis of decades of experience as a

- CS educator
- author
- algorithm designer
- software engineer
- Silicon Valley contributor
- CS researcher





Personal opinion . . . or unspoken consensus?

Fact of life in applied computing: performance matters

in a large number of important applications

Example: quadratic algorithms are
useless in modern applications

- millions or billions of inputs
- 10^{12} nanoseconds is 15+ minutes
- 10^{18} nanoseconds is 31+ years

- *Web commerce*
- *Bose-Einstein model*
- *String matching for genomics*
- *Natural language analysis*
- *N-body problem*

.
.
.
[long list]

Lessons:

1. Efficient algorithms enable solution of problems that could not otherwise be addressed.
2. Scientific method is essential in understanding program performance

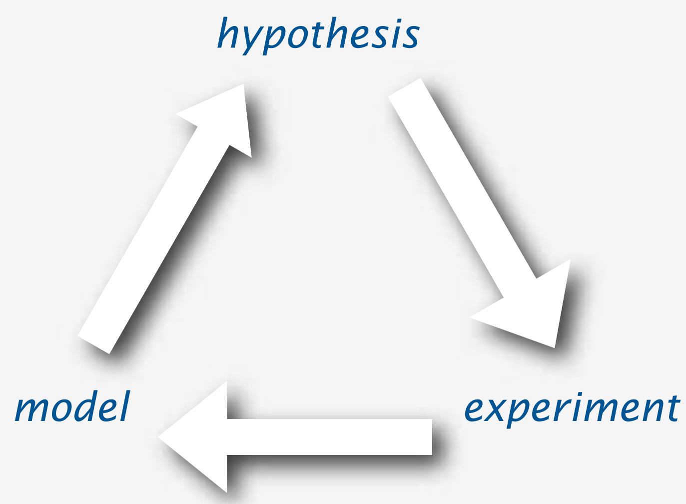
Important lessons for

- *beginners*
- *software engineers*
- *scientists*
- *[everyone]*

The scientific method

is essential in understanding program performance

Scientific method
- create a model describing natural world
- use model to develop hypotheses
- run experiments to validate hypotheses
- refine model and repeat



1950s: uses scientific method

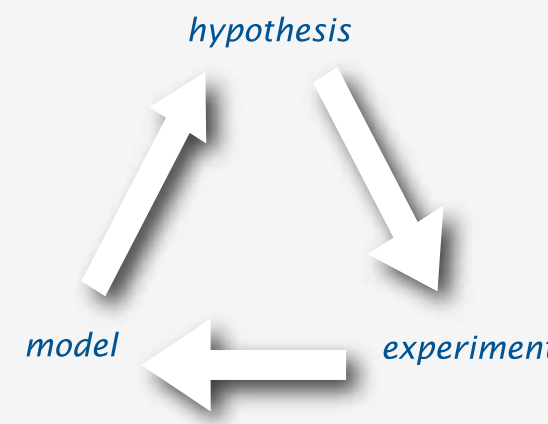
2000s: uses scientific method?

Algorithm designer who does not experiment gets lost in abstraction

Software developer who ignores cost risks catastrophic consequences

Warmup: random number generation

Problem: write a program to generate random numbers



model: classical probability and statistics

hypothesis: frequency values should be uniform

weak experiment:

- generate random numbers
- check for uniform frequencies

```
int k = 0;
while ( true )
  System.out.print(k++ % V);
```

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

random?

better experiment:

- generate random numbers
- use χ^2 test to check frequency values against uniform distribution

better hypotheses/experiments still needed

- many documented disasters
- active area of scientific research
- applications: simulation, cryptography
- connects to core issues in theory of computation

```
int k = 0;
while ( true )
{
   k = k*1664525 + 1013904223);
   System.out.print(k % V);
}
```

textbook algorithm that flunks χ^2 test

Warmup (continued)

Q. Is a given sequence of numbers random?

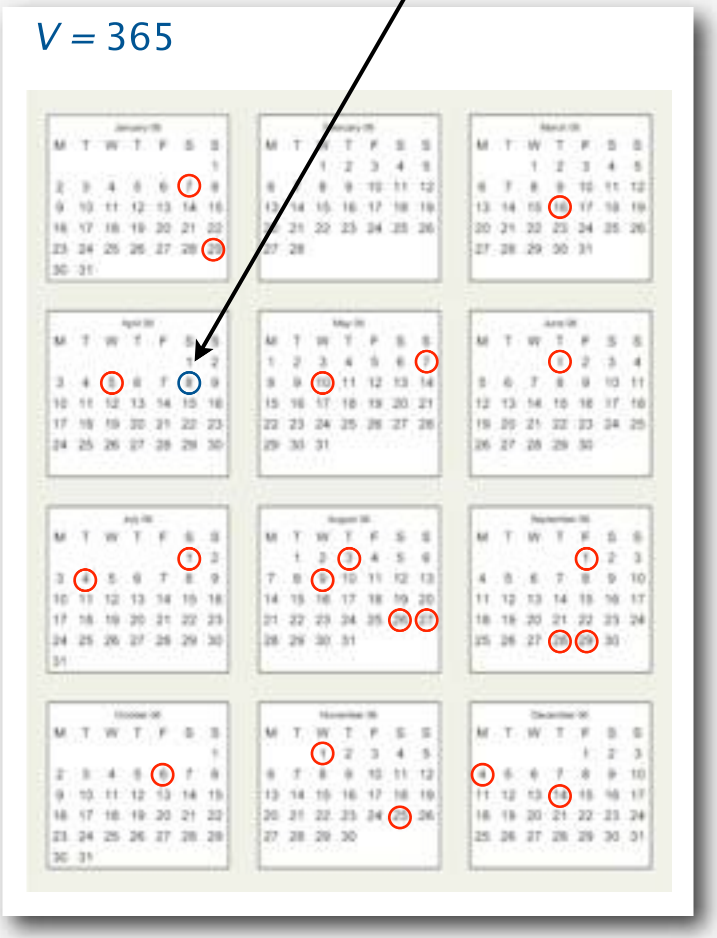
A. No.

Q. Does a given sequence exhibit some property that random number sequences exhibit?

Birthday paradox

Average count of random numbers generated until a duplicate happens is about $\sqrt{\pi V/2}$

average probes until duplicate is about 24



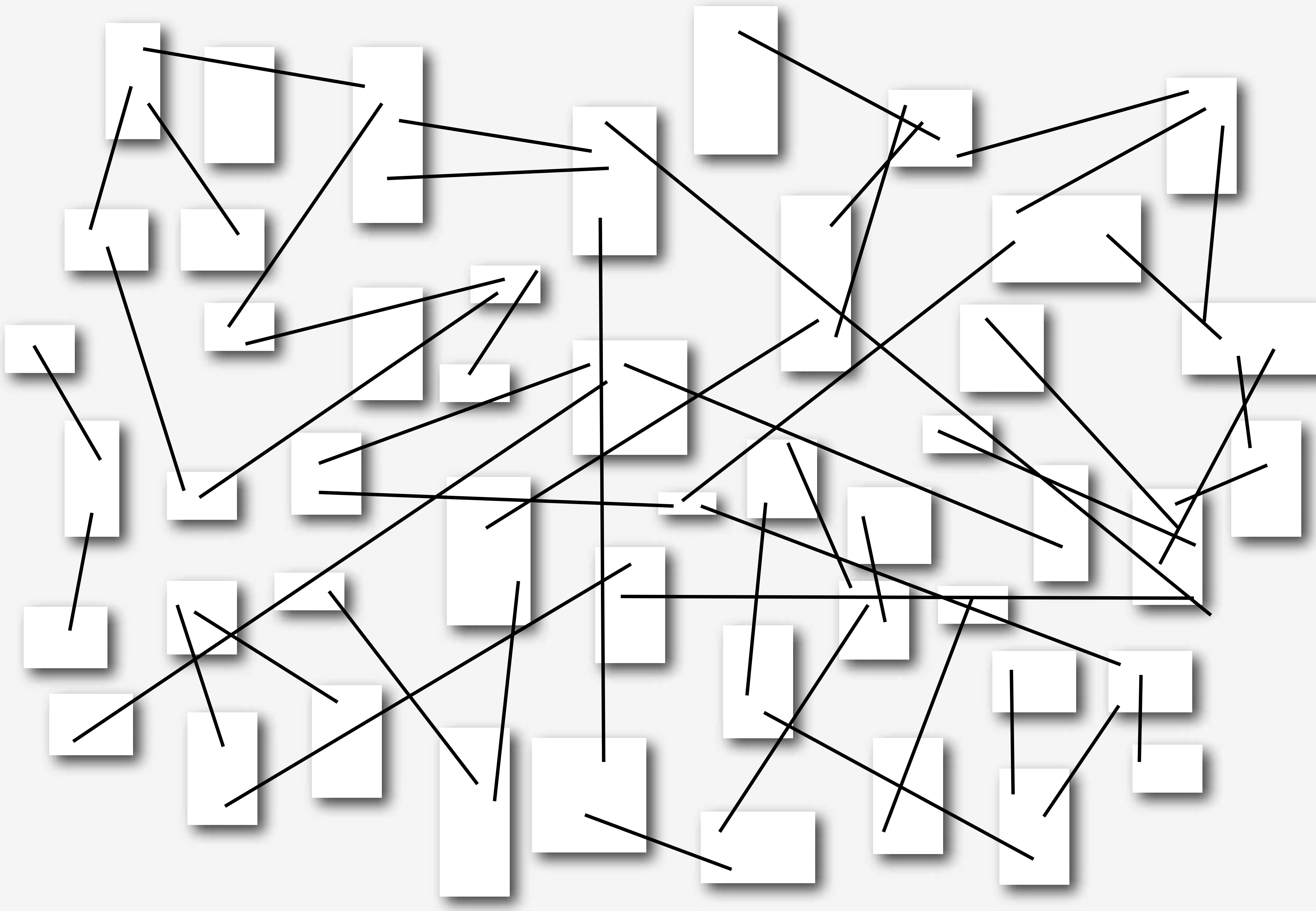
Example of a better experiment:

- generate numbers until duplicate
- check that count is close to $\sqrt{\pi V/2}$
- even better: repeat many times, check against distribution
- still better: run many similar tests for other properties

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" — John von Neumann

Preliminary hypothesis (needs checking)

Modern software requires huge amounts of code

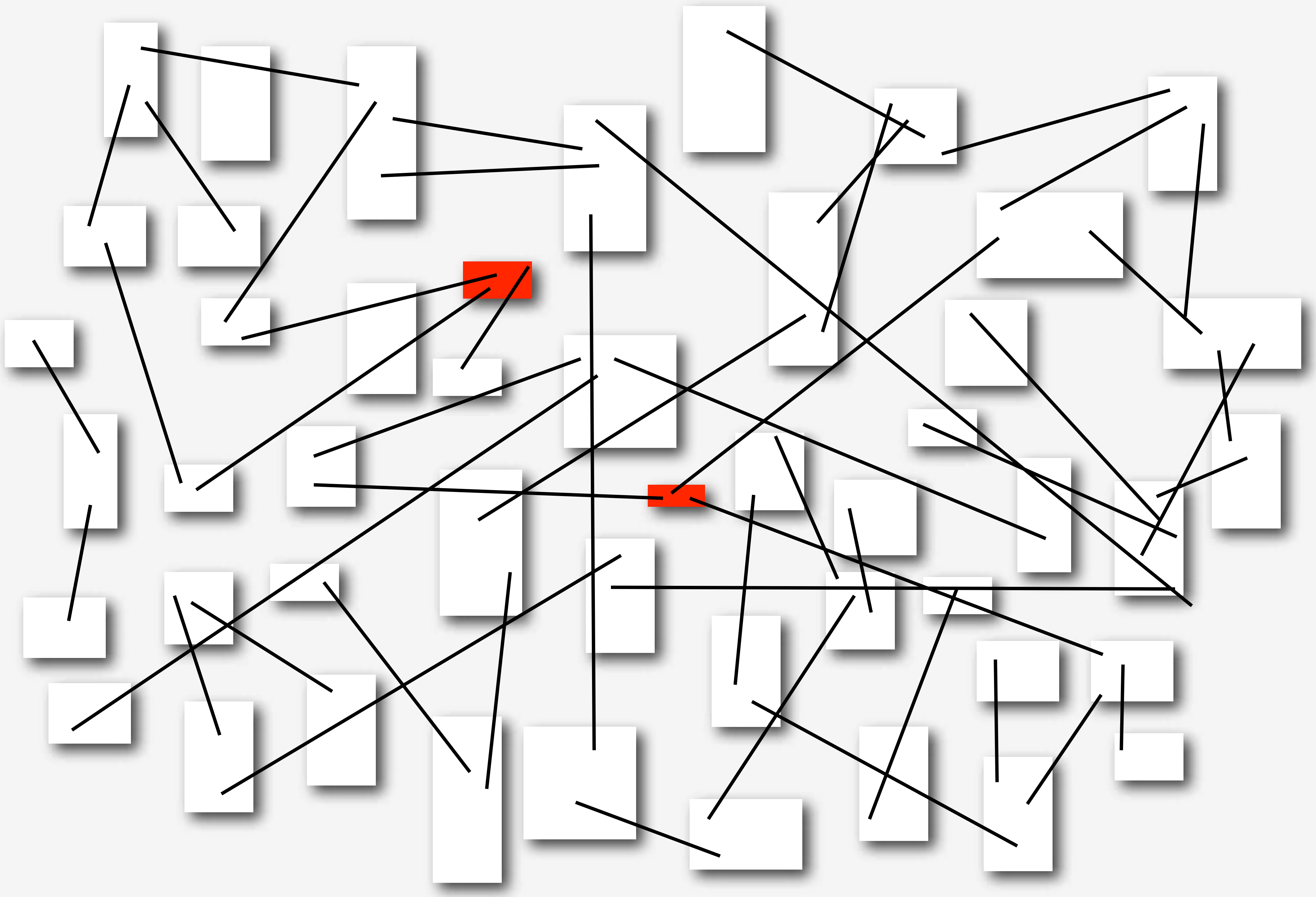


Preliminary hypothesis (needs checking)

Modern software development requires huge amounts of code

but

performance-critical code implements relatively few fundamental algorithms



Starting point

How to predict performance (to compare algorithms)?

Not the scientific method: O-notation

Theorem: Running time is $O(N^b)$

- hides details of implementation, properties of input
- useful for classifying algorithms and complexity classes
- not at all useful for predicting performance

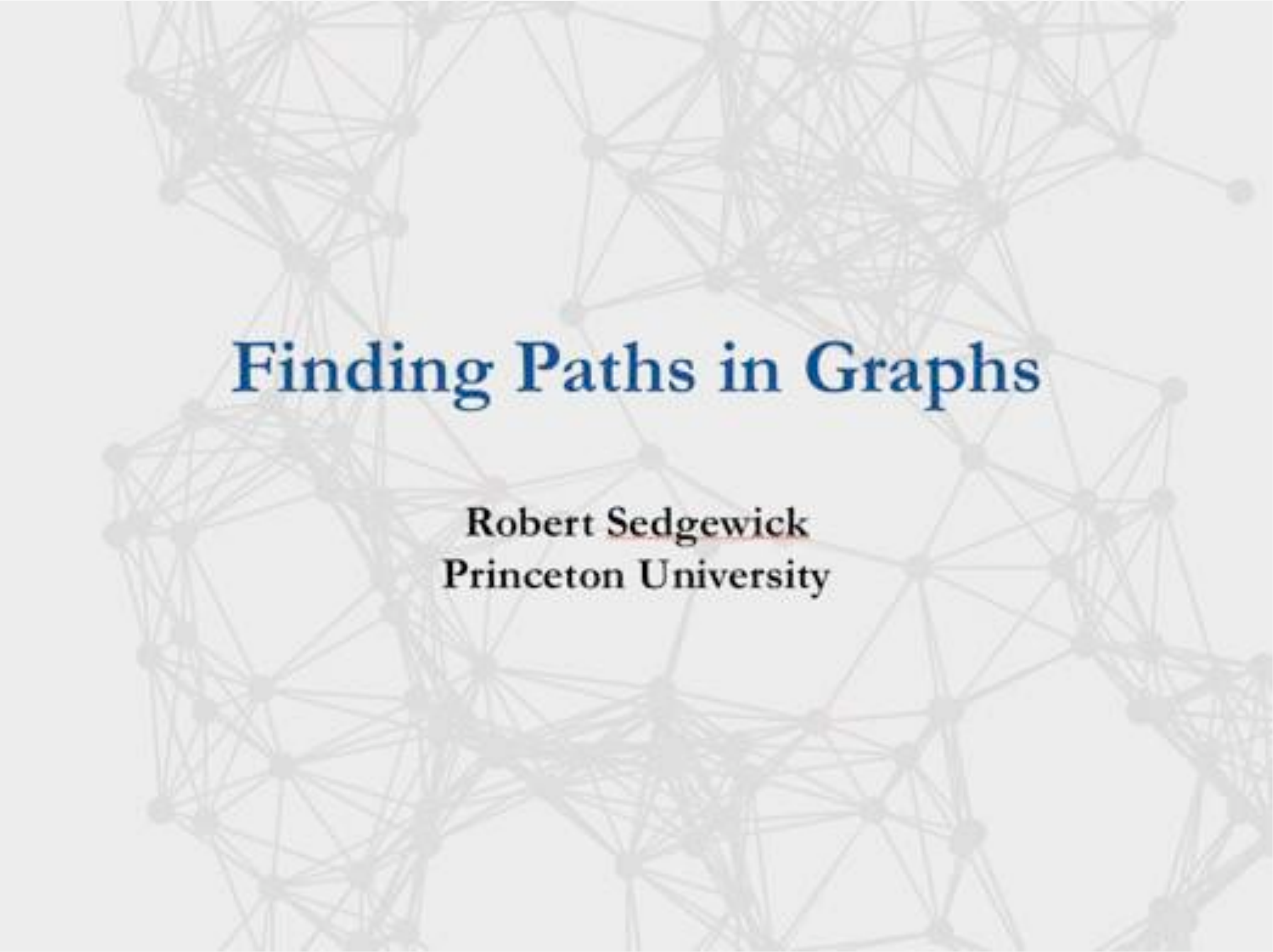
Scientific method: Tilde-notation.

Hypothesis: Running time is $\sim aN^b$

- doubling test: $T(2N)/T(N) \sim 2^b$
- an effective way to predict performance

Detailed example: paths in graphs

A lecture within a lecture



Finding an *st*-path in a graph

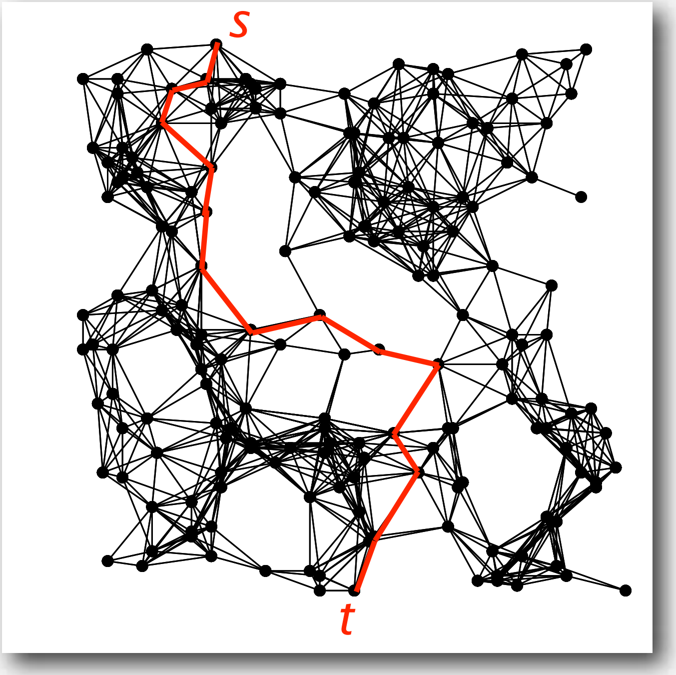
is a fundamental operation that demands understanding

Ground rules for this talk

- work in progress (more questions than answers)
- basic research
- save “deep dive” for the right problem

Applications

- graph-based optimization models
- networks
- percolation
- computer vision
- social networks
- (many more)



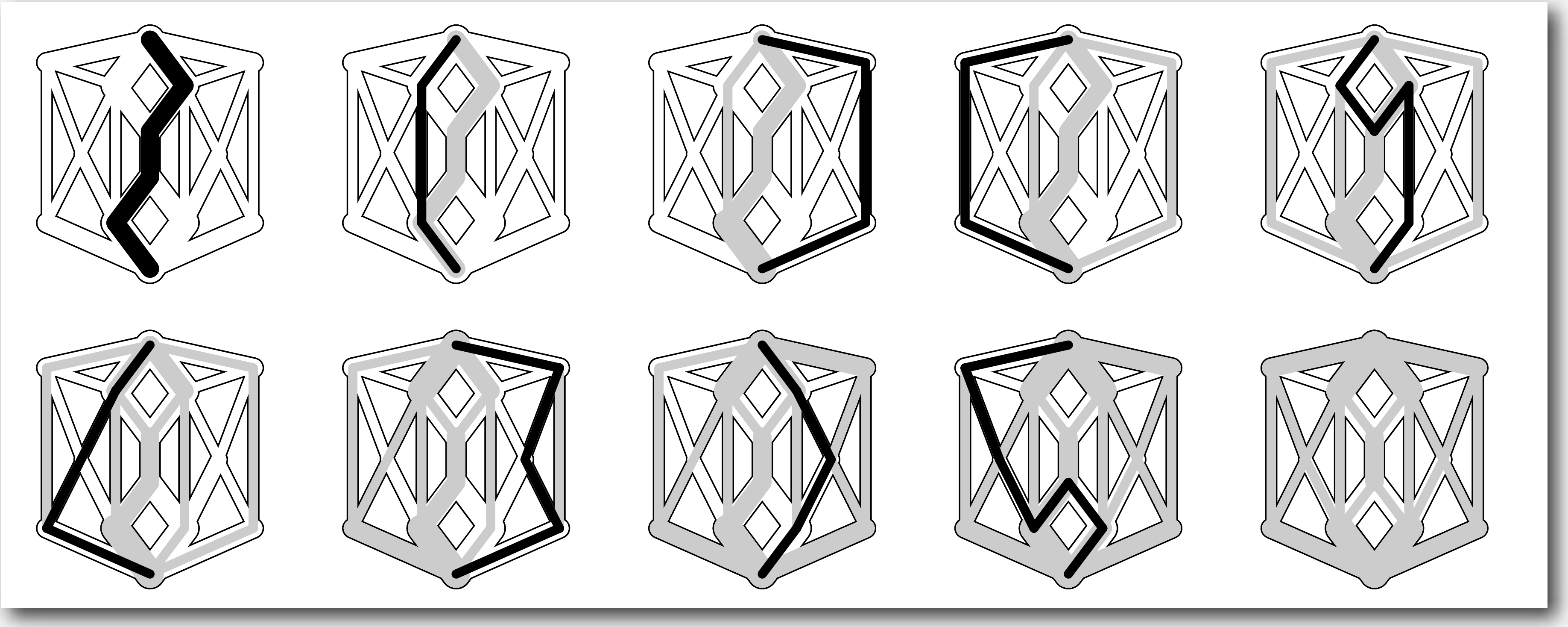
Basic research

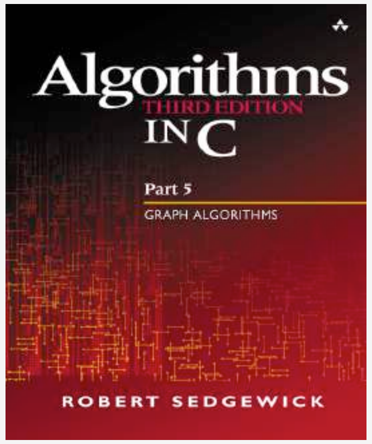
- fundamental abstract operation with numerous applications
- worth doing even if no immediate application
- resist temptation to prematurely study impact

Motivating example: maxflow

Ford-Fulkerson maxflow scheme

- find any s-t path in a (residual) graph
- augment flow along path (may create or delete edges)
- iterate until no path exists





Goal: compare performance of two basic implementations

- shortest augmenting path
- maximum capacity augmenting path

Key steps in analysis

- How many augmenting paths?
- What is the cost of finding each path?

research literature

this talk

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters

- number of vertices V
- number of edges E
- maximum capacity C

How many augmenting paths?

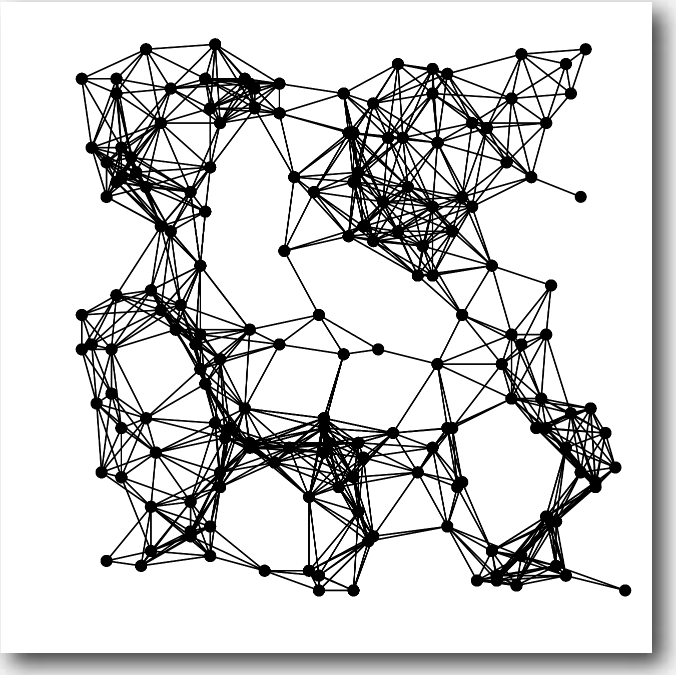
	worst case upper bound
shortest	VE/2 VC
max capacity	2E lg C

How many steps to find each path? E (worst-case upper bound)

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path



Graph parameters for example graph

- number of vertices V = 177
- number of edges E = 2000
- maximum capacity C = 100

How many augmenting paths?

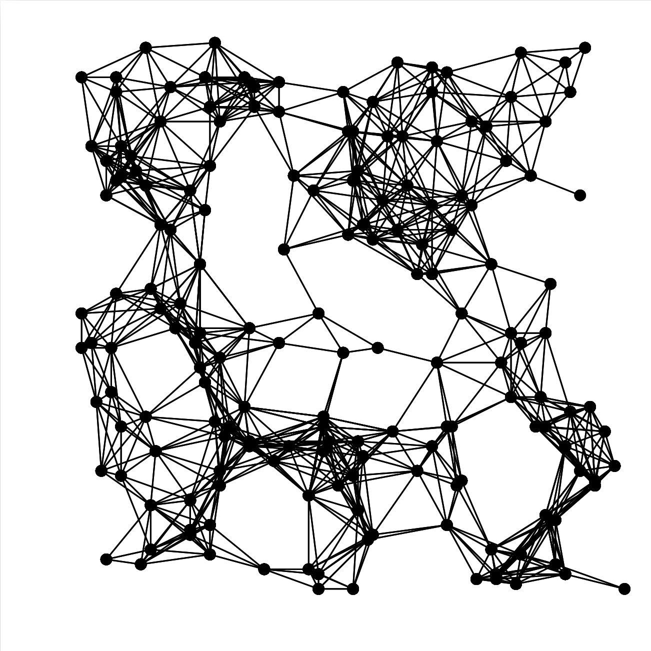
	worst case upper bound	*for example*
shortest	VE/2 VC	177,000 17,700
max capacity	2E lg C	26,575

How many steps to find each path? 2000 (worst-case upper bound)

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path



Graph parameters for example graph

- number of vertices V = 177
- number of edges E = 2000
- maximum capacity C = 100

How many augmenting paths?

	worst case upper bound	for example	actual
shortest	VE/2 VC	177,000 17,700	37
max capacity	2E lg C	26,575	7

How many steps to find each path? < 20, on average

total is a factor of 1 million high for thousand-node graphs!

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters

- number of vertices V
- number of edges E
- maximum capacity C

Total number of steps?

	worst case upper bound
shortest	$VE^2/2$ VEC
max capacity	$2E^2 \lg C$

WARNING: The Algorithm General has determined that using such results to predict performance or to compare algorithms may be hazardous.

Motivating example: lessons

Goals of algorithm analysis

- predict performance (running time)
- guarantee that cost is below specified bounds

Common wisdom

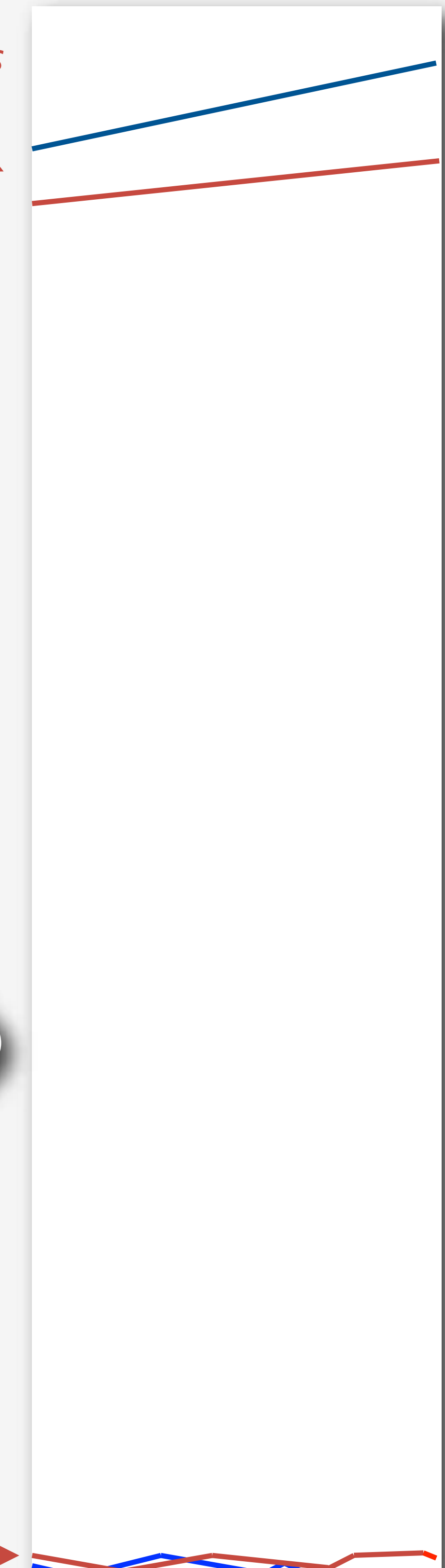
- random graph models are unrealistic
- average-case analysis of algorithms is too difficult
- worst-case performance bounds are the standard

Unfortunate truth about worst-case bounds

- often useless for prediction (fictional)
- often useless for guarantee (too high)
- often misused to compare algorithms

Bounds are useful in some applications:

Open problem: Do better!



Surely, we can do better

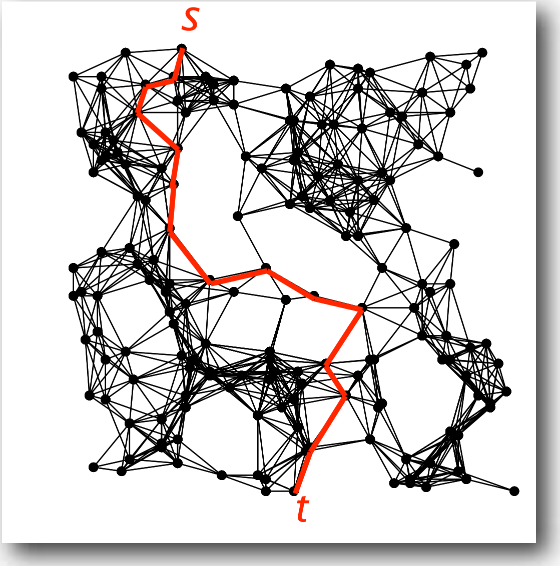
An actual exchange with a theoretical computer scientist:

TCS (in a talk):	Algorithm A is bad. Google should be interested in my new Algorithm B.
RS:	What's the matter with Algorithm A?
TCS:	It is not optimal. It has an extra O(log log N) factor.
RS:	But Algorithm B is very complicated, lg lg N is less than 6 in this universe, and that is just an upper bound. Algorithm A is certainly going to run 10 to 100 times faster in any conceivable real-world situation. Why should Google care about Algorithm B?
TCS:	Well, I like it. I don't care about Google.

Finding an *st*-path in a graph

is a basic operation in a great many applications

Q. What is the best way to find an *st*-path in a graph?



A. Several well-studied textbook algorithms are known

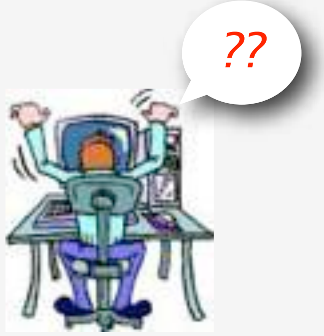
- Breadth-first search (BFS) finds the shortest path
- Depth-first search (DFS) is easy to implement
- Union-Find (UF) needs two passes

BUT

- all three process all E edges in the worst case
- diverse kinds of graphs are encountered in practice

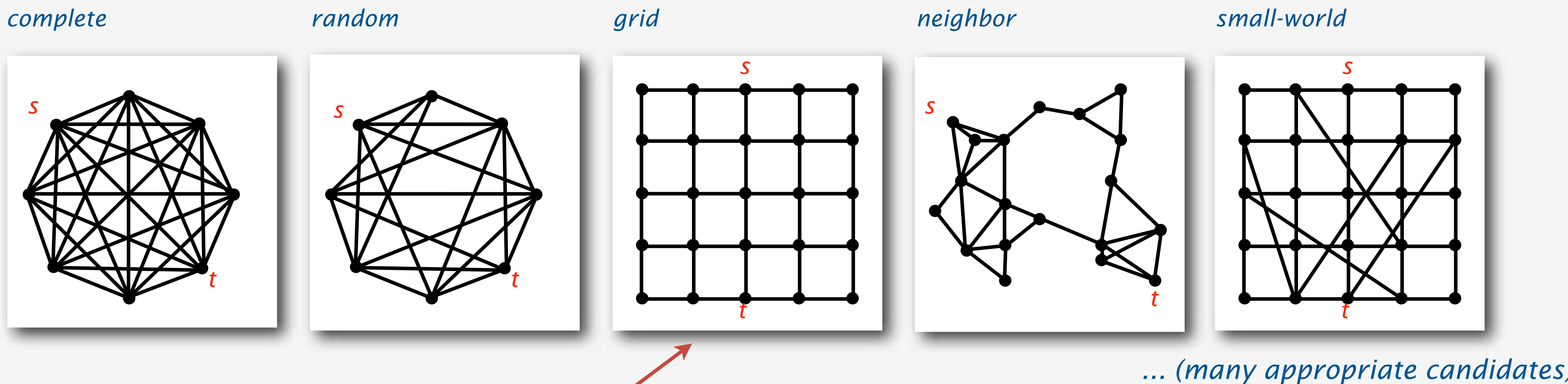
Worst-case analysis is useless for predicting performance

Which basic algorithm should a practitioner use?



Grid graphs

Algorithm performance depends on the graph model



Initial choice: grid graphs

- sufficiently challenging to be interesting
- found in practice (or similar to graphs found in practice)
- scalable
- potential for analysis

Ex: easy to find short paths quickly with A in geometric graphs (stay tuned)*

Ground rules

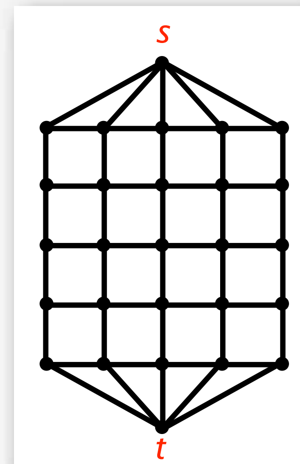
- algorithms should work for all graphs
- algorithms should not use any special properties of the model

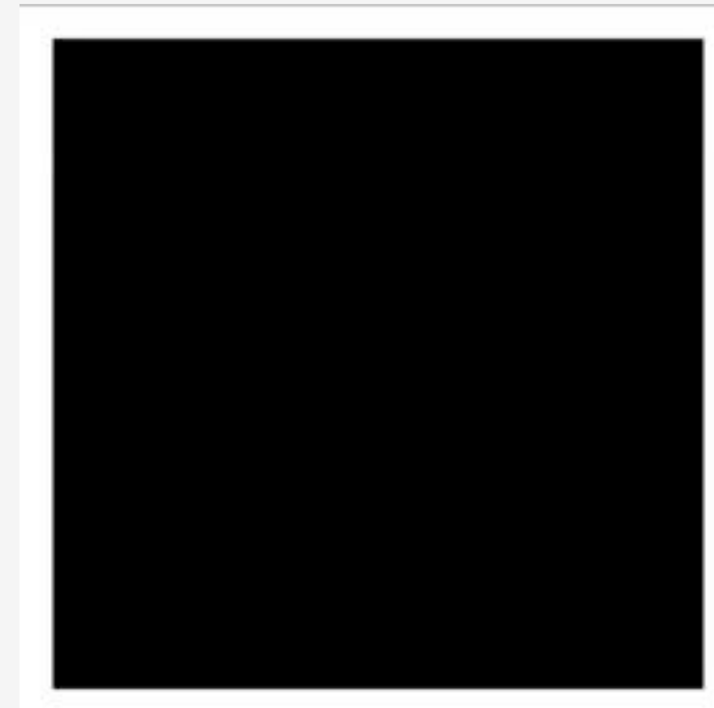
Applications of grid graphs

conductivity
concrete
granular materials
porous media
polymers
forest fires
epidemics
Internet
resistor networks
evolution
social influence
Fermi paradox
fractal geometry
stereo vision
image restoration
object segmentation
scene reconstruction
.
.
.

Example 1: Percolation

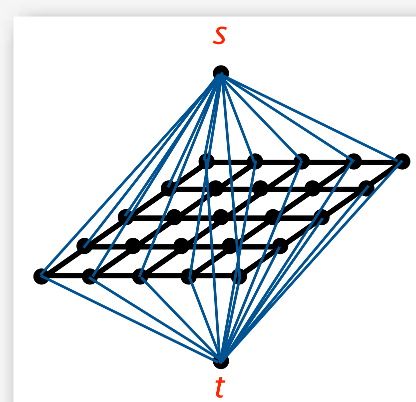
- widely-studied model
- few answers from analysis
- arbitrarily huge graphs





Example 2: Image processing

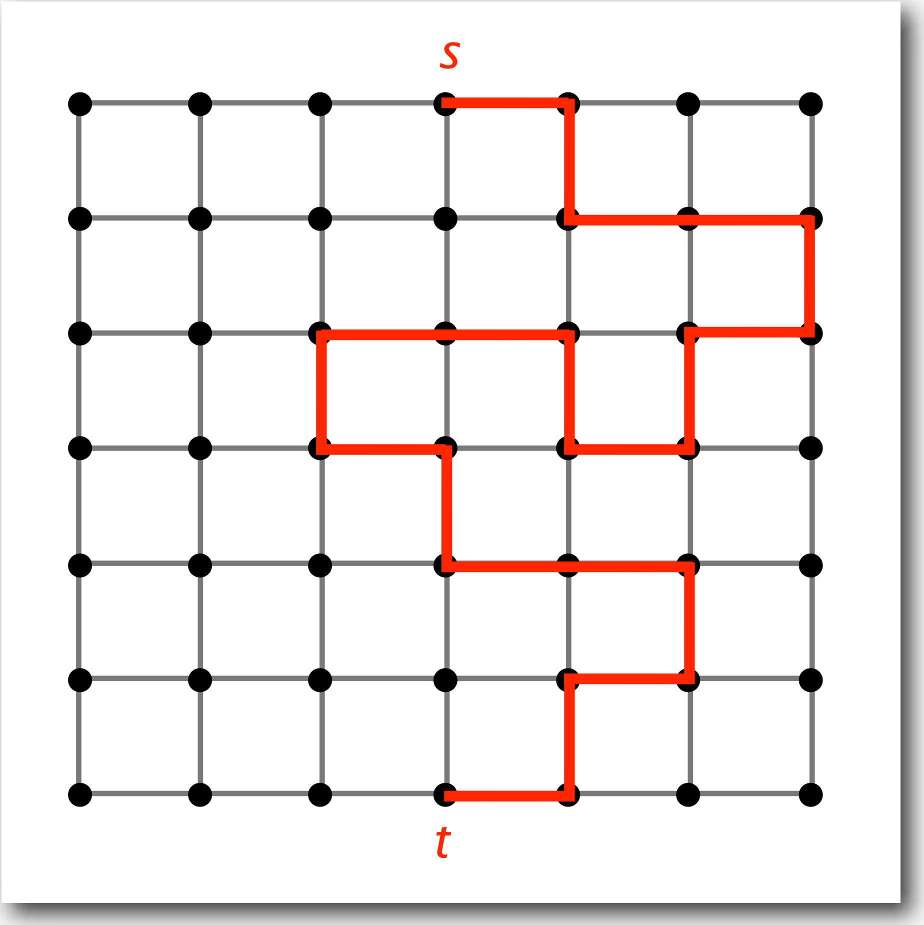
- model pixels in images
- DFS, maxflow/mincut, and other algs
- huge graphs



Finding an *st*-path in a grid graph

M by M grid of vertices
undirected edges connecting each vertex to its HV neighbors
source vertex s at center of top boundary
destination vertex t at center of bottom boundary

Find any path connecting s to t



M^2 vertices
about $2M^2$ edges

M	vertices	edges
7	49	84
15	225	420
31	961	1860
63	3969	7812
127	16129	32004
255	65025	129540
511	261121	521220

Cost measure: number of graph edges examined

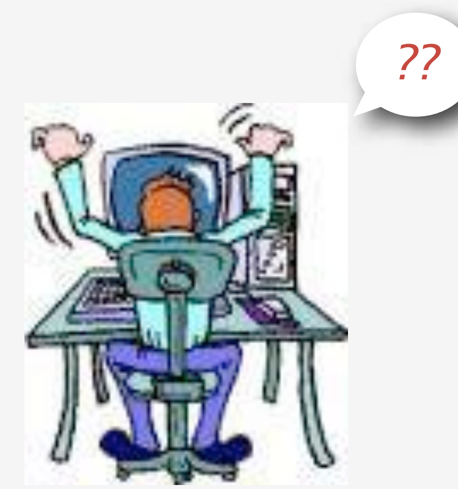
Finding an *st*-path in a grid graph

Similar problems are covered extensively in the literature

- Percolation
- Random walk
- Nonselfintersecting paths in grids
- Graph covering
- . . .

Elementary algorithms are found in textbooks

- Depth-first search (DFS)
- Breadth-first search (BFS)
- Union-find
- . . .



Which basic algorithm should a practitioner use to find a path in a grid-like graph?

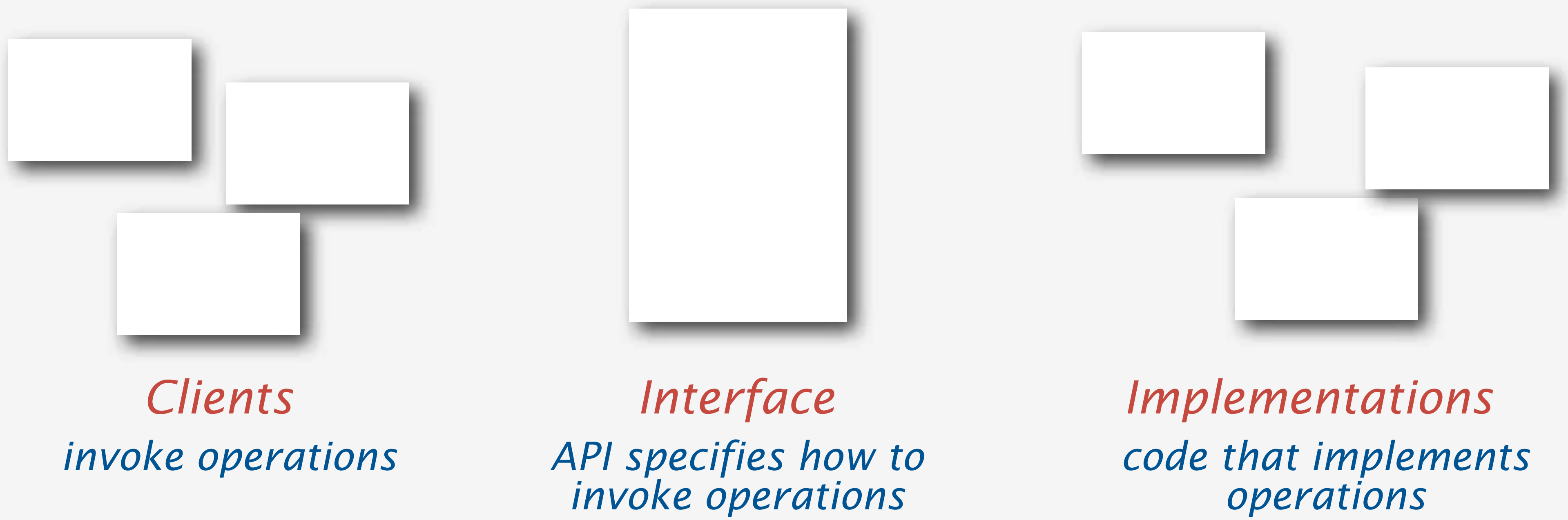
Literature is no help, so

- Implement elementary algorithms
- Use scientific method to study performance

Data abstraction

a modern tool to separate clients from implementations

A data type is a set of values and the operations performed on them
An abstract data type (ADT) is a data type whose representation is hidden
An applications programming interface (API) is a specification



Implementation should not be tailored to particular client

Develop implementations that work properly for all clients
Study their performance for the client at hand

Implementing a GRAPH data type

is an exercise in software engineering

Sample "design pattern" (for this talk)

Vertices are integers in [0, V)
Edges are vertex pairs

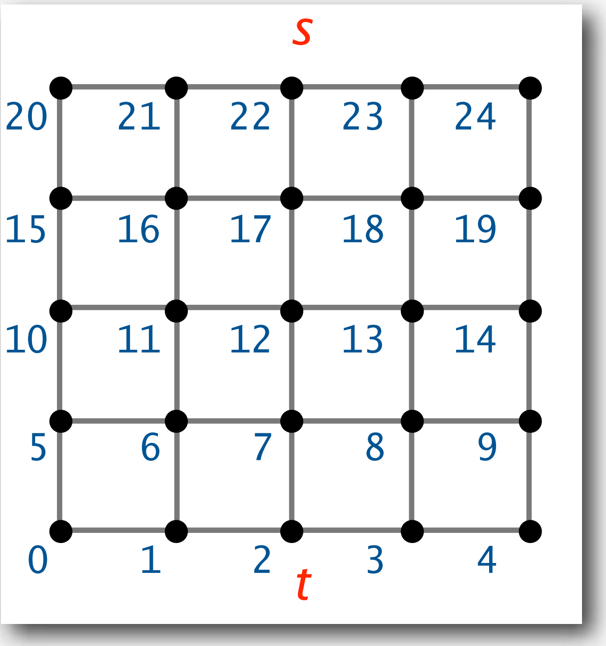
GRAPH *API*

```
public class GRAPH
        GRAPH(Edge[] a)             construct a GRAPH from an array of edges
   void findPath(int s, int t)      conduct a search from s to t
    int st(int v)                   return predecessor of v on path found
```

Client code for grid graphs

```
int e = 0;
Edge[] a = new Edge[E];
for (int i = 0; i < V; i++)
 {  if (i < V-M) a[e++] = new Edge(i, i+M);
    if (i >= M) a[e++] = new Edge(i, i-M);
    if ((i+1) % M != 0) a[e++] = new Edge(i, i+1);
    if (i % M != 0) a[e++] = new Edge(i, i-1);
 }
GRAPH G = new GRAPH(a);
G.findPath(V-1-M/2, M/2);
for (int k = t; k != s; k = G.st(k))
  System.out.println(s + "-" + t);
```

M = 5



Three standard ways to find a path

Depth-first search (DFS): recursive (stack-based) search

Breadth-first search (BFS): queue-based shortest-path search

Union-find (UF): use classic set-equivalence algorithms

DFS

```
DFS(s)
--------------------
    DFS(v):
      done if v = t
      if v unmarked
        mark v
        DFS(v)
```

BFS

```
put s on Q
while Q is nonempty
  get x from Q
  done if x = t
  for each v adj to x
    if v unmarked
      put v on Q
      mark v
```

UF

```
for each edge u-v
  union (u, v)
  done if s and t are
  in the same set
--------------------
run DFS or BFS on set
containing s and t
```

First step: Implement GRAPH using each algorithm

Depth-first search: a standard implementation

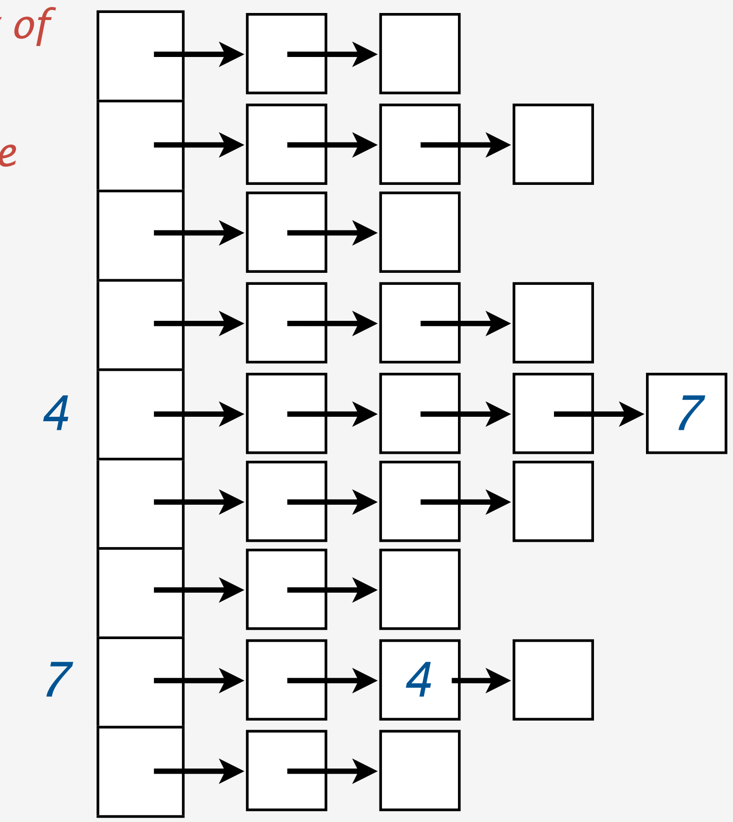
GRAPH constructor code

```
for (int k = 0; k < E; k++)
  { int v = a[k].v, w = a[k].w;
    adj[v] = new Node(w, adj[v]);
    adj[w] = new Node(v, adj[w]);
  }
```

graph representation

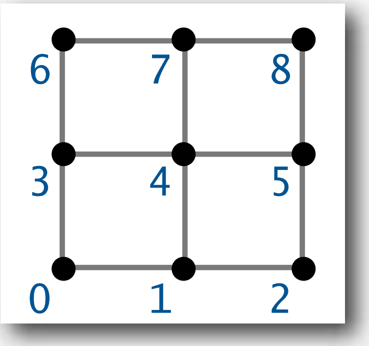
vertex-indexed array of linked lists

two nodes per edge



DFS implementation (code to save path omitted)

```
void findPathR(int s, int t)
  { if (s == t) return;
    visited(s) = true;
    for(Node x = adj[s]; x != null; x = x.next)
      if (!visited[x.v]) findPathR(x.v, t);
  }
void findPath(int s, int t)
  { visited = new boolean[V];
    searchR(s, t);
  }
```



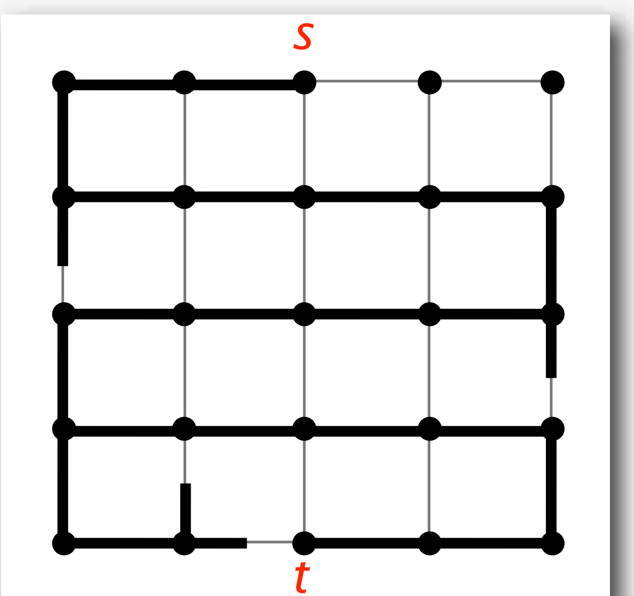
Basic flaw in standard DFS scheme

cost strongly depends on arbitrary decision in client (!!)

```
...
for (int i = 0; i < V; i++)
  {
    if ((i+1) % M != 0) a[e++] = new Edge(i, i+1);
    if (i % M != 0) a[e++] = new Edge(i, i-1);
    if (i < V-M) a[e++] = new Edge(i, i+M);
    if (i >= M) a[e++] = new Edge(i, i-M);
  }
...
```

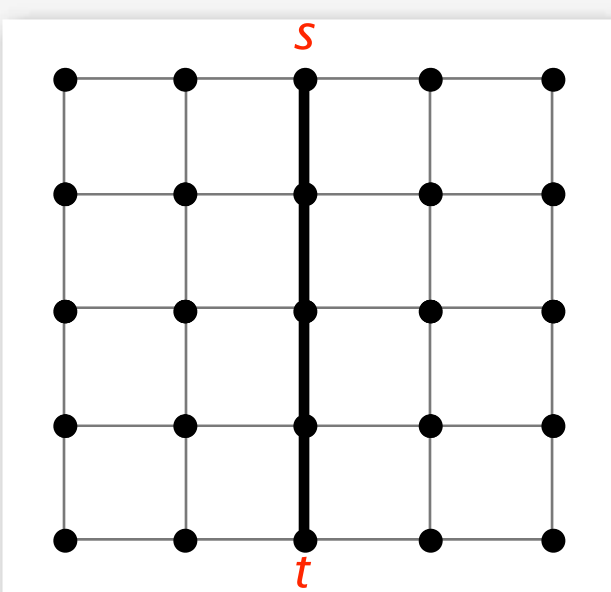
order of these statements determines order in lists

west, east, north, south



~E/2

south, north, east, west



~E $^{1/2}$

order in lists has drastic effect on running time

bad news for ANY graph model

Addressing the basic flaw

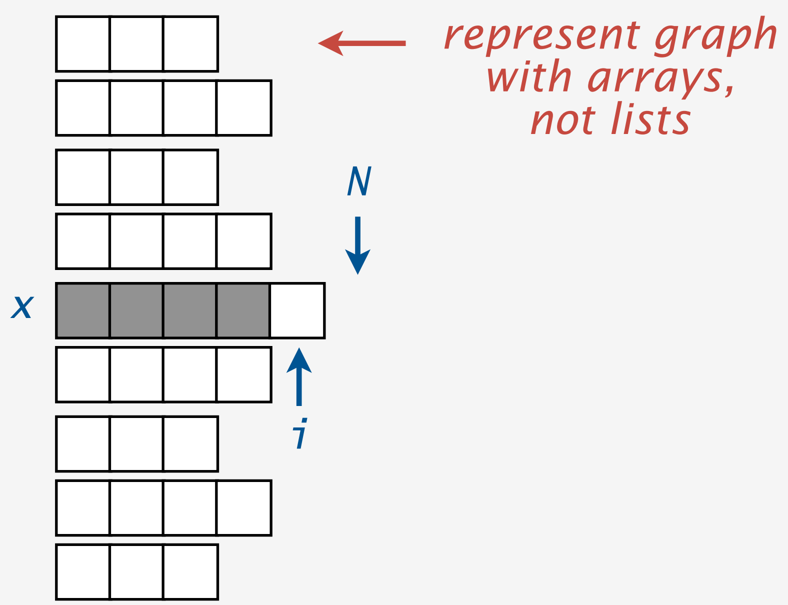
Advise the client to randomize the edges?

- no, very poor software engineering
- leads to nonrandom edge lists (!)

Randomize each edge list before use?

- no, may not need the whole list

Solution: Use a randomized iterator



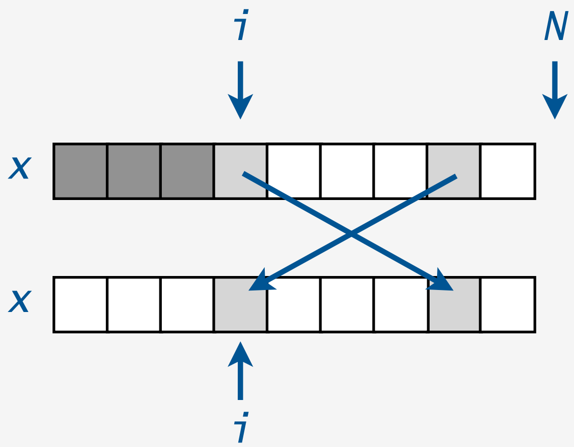
standard iterator

```
int N = adj[x].length;
for(int i = 0; i < N; i++)
  { process vertex adj[x][i]; }
```

randomized iterator

```
int N = adj[x].length;
for(int i = 0; i < N; i++)
  { exch(adj[x], i, i + (int) Math.random()*(N-i));
    process vertex adj[x][i];
  }
```

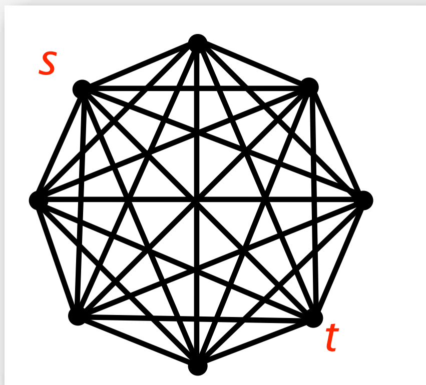
exchange random vertex from `adj[x][i..N-1]` *with* `adj[x][i]`



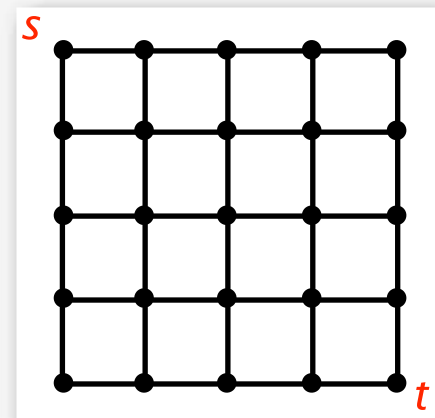
Use of randomized iterators

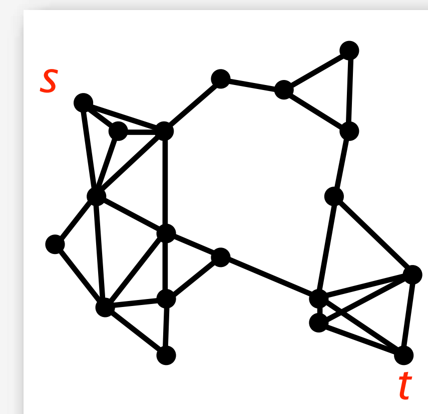
turns every graph algorithm into a randomized algorithm

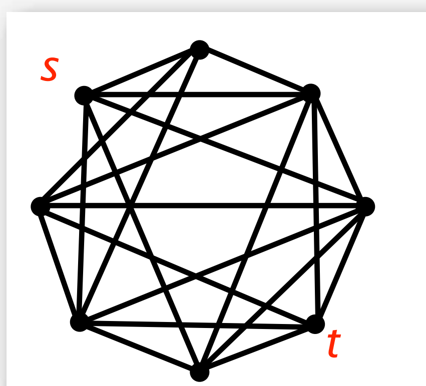
Important practical effect: stabilizes algorithm performance

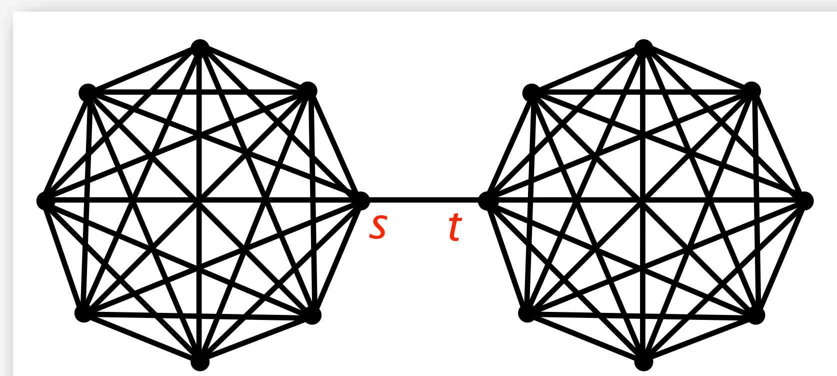


cost depends on problem
not its representation









Yields well-defined and fundamental analytic problems

- Average-case analysis of algorithm X for graph family Y(N)?
- Distributions?
- Full employment for algorithm analysts

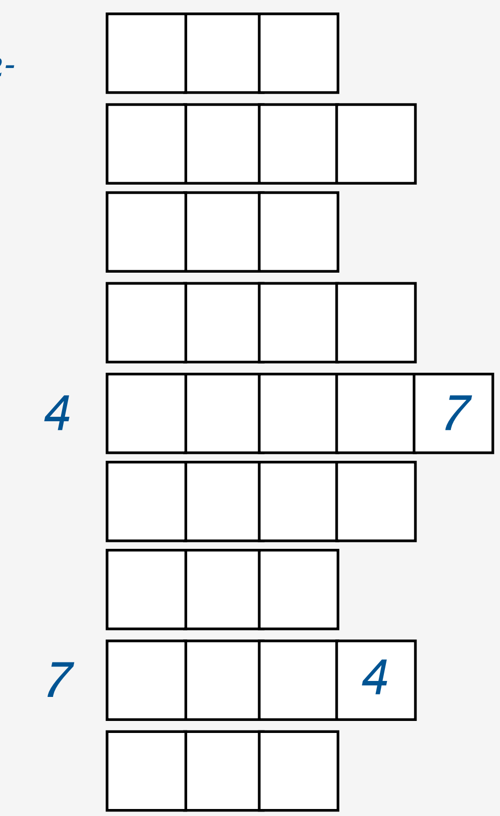
(Revised) standard DFS implementation

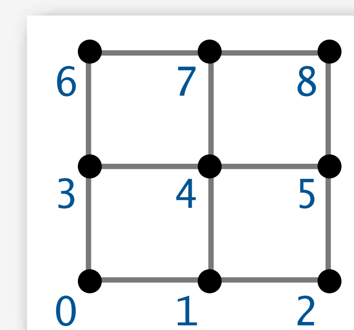
graph ADT constructor code

```
for (int k = 0; k < E; k++)
  {
    int v = a[k].v, w = a[k].w;
    adj[v][deg[v]++] = w;
    adj[w][deg[w]++] = v;
  }
```

graph representation

vertex-indexed array of variable-length arrays





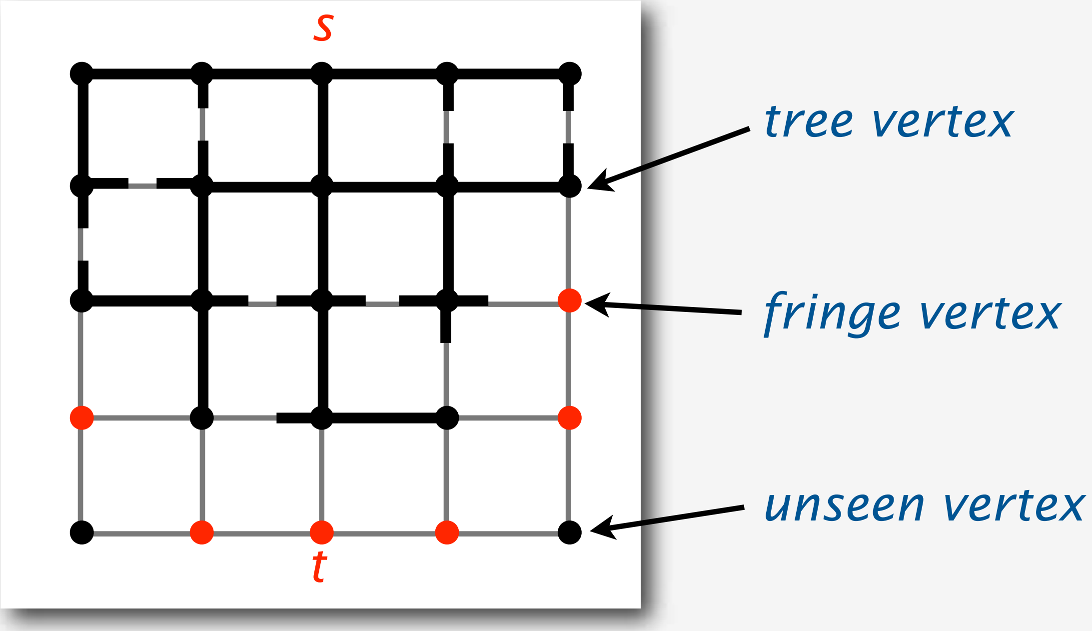
DFS implementation (code to save path omitted)

```
void findPathR(int s, int t)
  {
    int N = adj[s].length;
    if (s == t) return;
    visited(s) = true;
    for(int i = 0; i < N; i++)
    {
      int v = exch(adj[s], i, i+(int) Math.random()*(N-i));
      if (!visited[v]) searchR(v, t);
    }
  }
void findPath(int s, int t)
  {
    visited = new boolean[V];
    findpathR(s, t);
  }
```

BFS: standard implementation

Use a queue to hold fringe vertices

put s on Q
while Q is nonempty
get x from Q
done if x = t
for each unmarked v adj to x
put v on Q
mark v




```
void findPath(int s, int t)
 { Queue Q = new Queue();
   Q.put(s); visited[s] = true;
   while (!Q.empty())
     { int x = Q.get(); int N = adj[x].length;
       if (x == t) return;
       for (int i = 0; i < N; i++)
         { int v = exch(adj[x], i, i + (int) Math.random()*(N-i));
           if (!visited[v])
             { Q.put(v); visited[v] = true; }          }
     }
 }
```

FIFO queue for BFS

randomized iterator

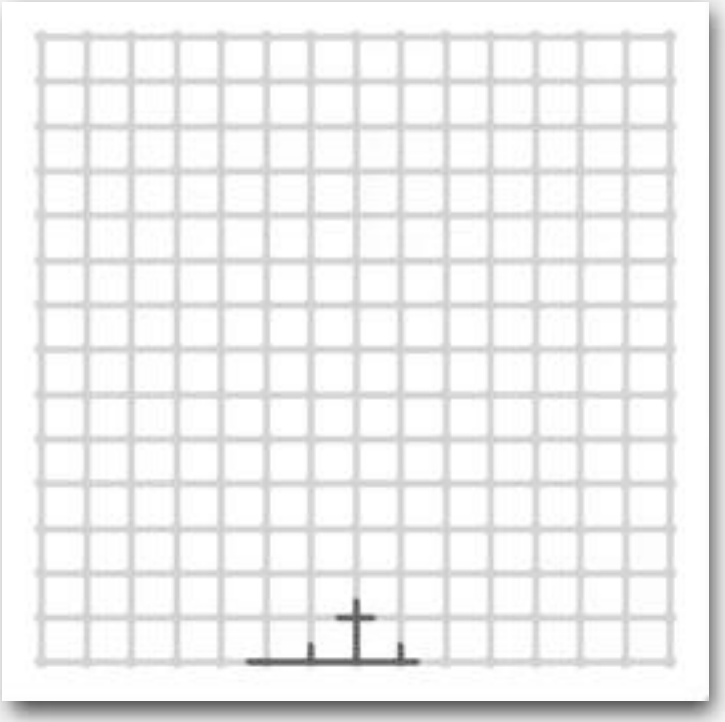
Generalized graph search: other queues yield DFS, A* and other algorithms

Animations

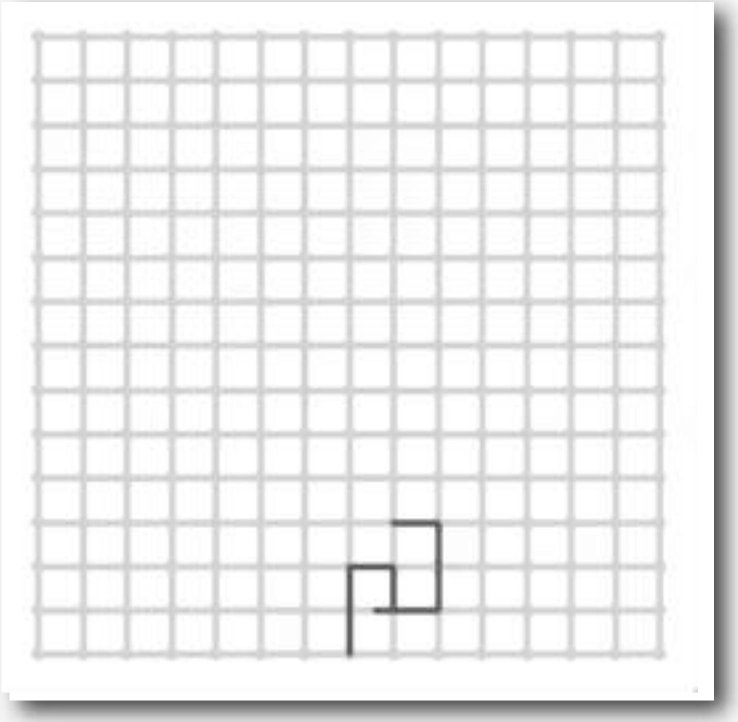
give intuition on performance
and suggest hypotheses to verify with experimentation

Aside: Are you using animations like this regularly?
Why not?

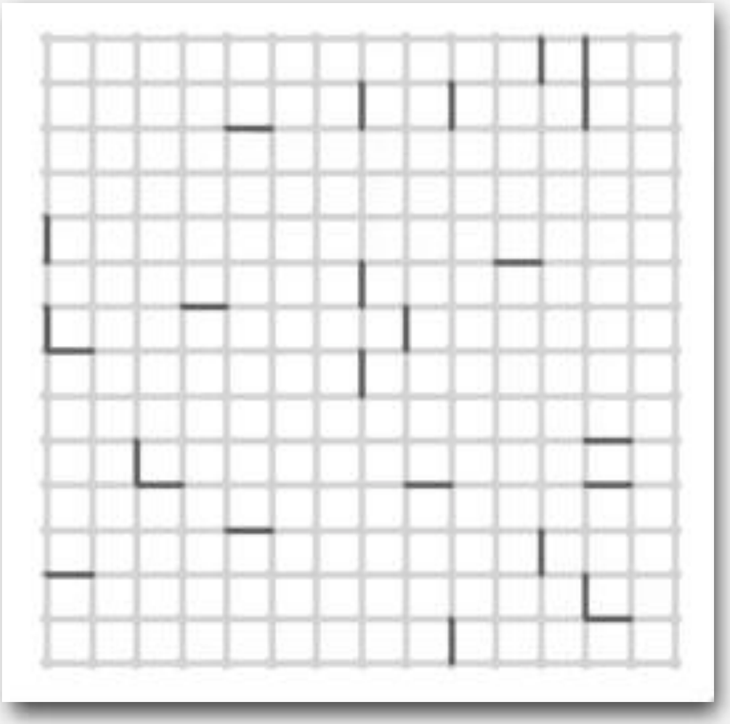
BFS



DFS



UF (code omitted)



Experimental results

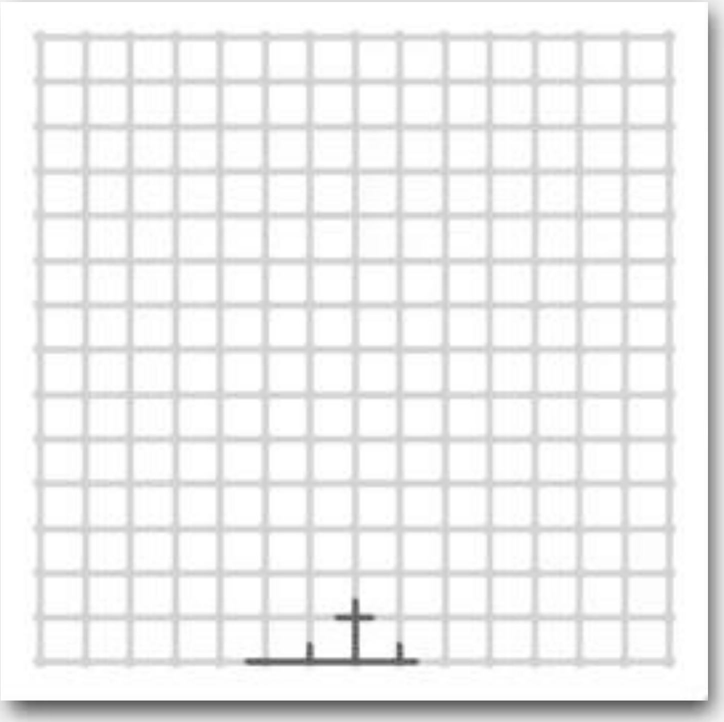
show that DFS is faster than BFS and UF on the average

M	V	E	BFS	DFS	UF
7	49	168	0.75	0.32	1.05
15	225	840	0.75	0.45	1.02
31	961	3720	0.75	0.36	1.14
63	3969	15624	0.75	0.32	1.05
127	16129	64008	0.75	0.40	0.99
255	65025	259080	0.75	0.42	1.08

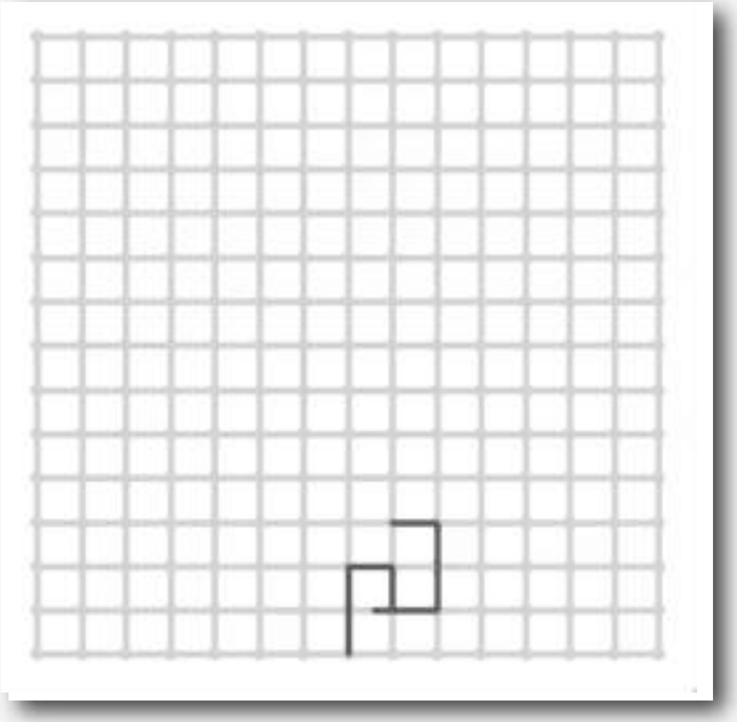
Analytic proof?

Faster algorithms available?

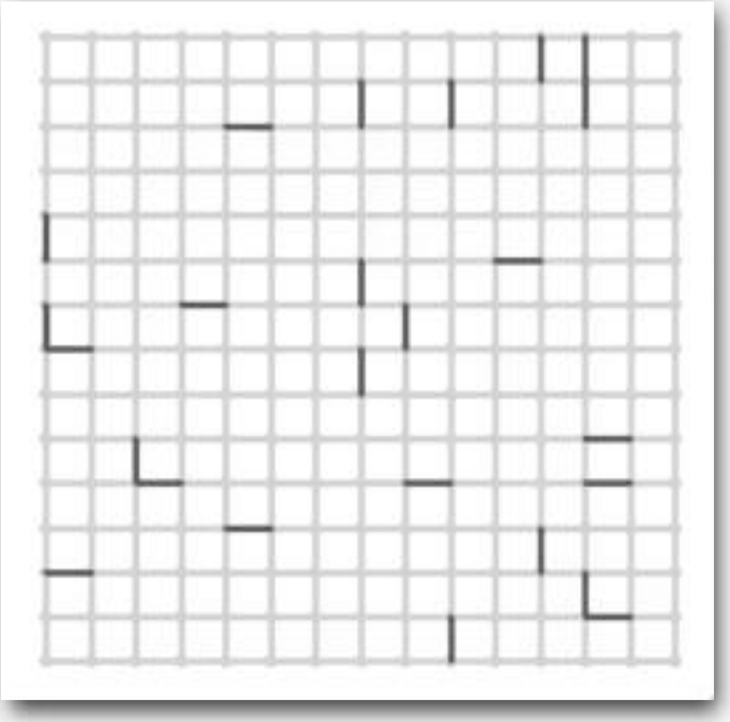
BFS



DFS



UF



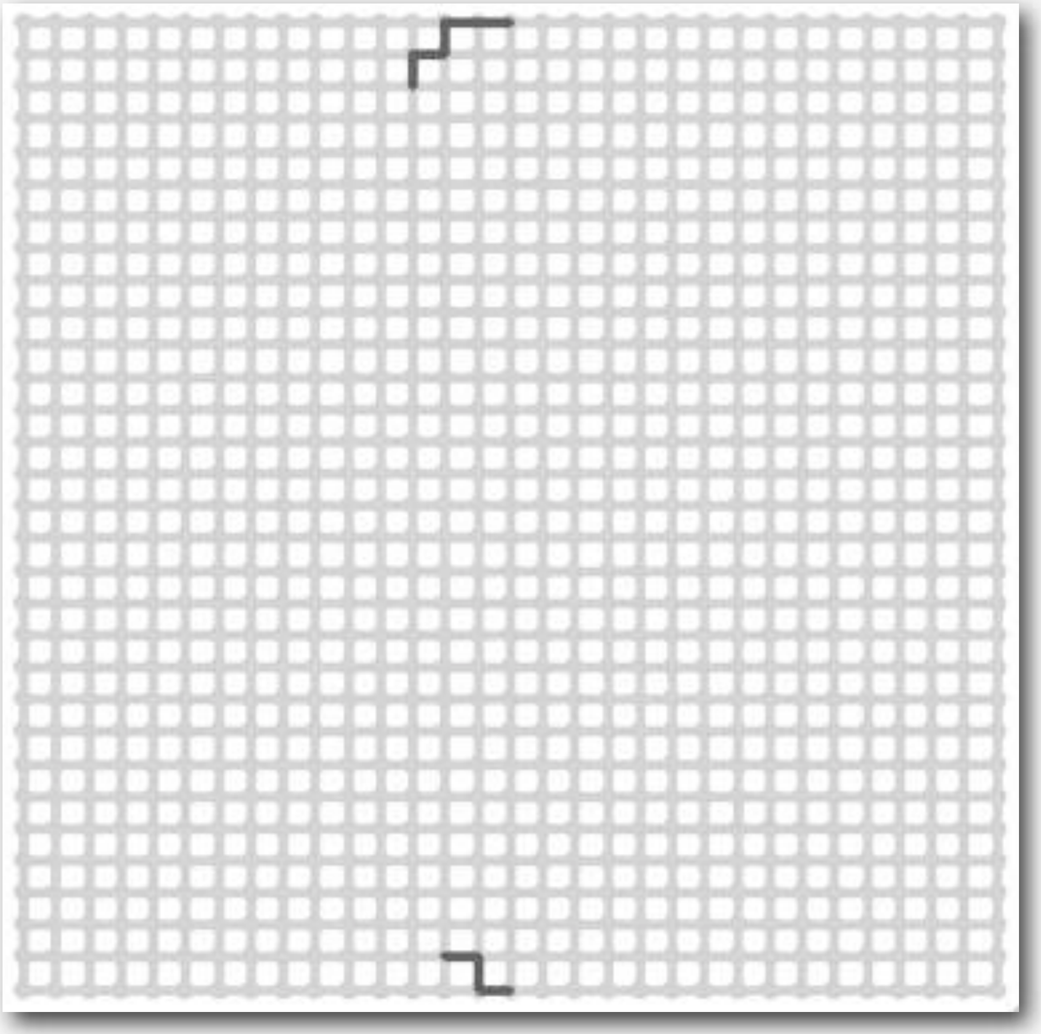
A standard search paradigm

gives a faster algorithm or finding an *st*-path in a graph

Use two depth-first searches

- one from the source
- one from the destination
- interleave the two

M	V	E	BFS	DFS	UF	two
7	49	168	0.75	0.32	1.05	0.18
15	225	840	0.75	0.45	1.02	0.13
31	961	3720	0.75	0.36	1.14	0.15
63	3969	15624	0.75	0.32	1.05	0.14
127	16129	64008	0.75	0.40	0.99	0.13
255	65025	259080	0.75	0.42	1.08	0.12



Examines 13% of the edges

3-8 times faster than standard implementations

Not log log N, but not bad!

Faster approach?

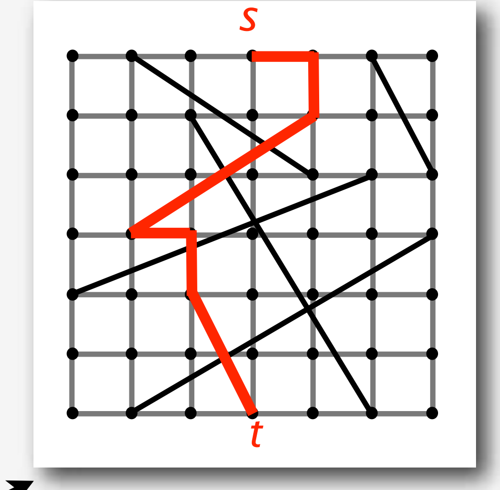
Other models?

Small-world graphs

are a widely studied graph model with many applications

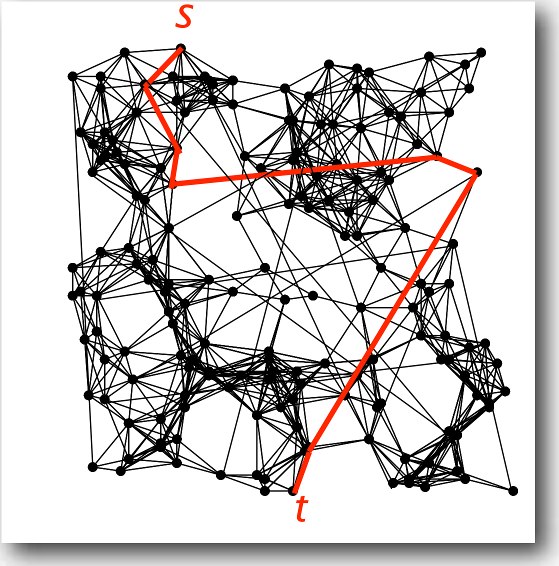
A small-world graph has

- large number of vertices
- low average vertex degree (sparse)
- low average path length
- local clustering



Examples:

- Add random edges to grid graph
- Add random edges to any sparse graph with local clustering
- Many scientific models



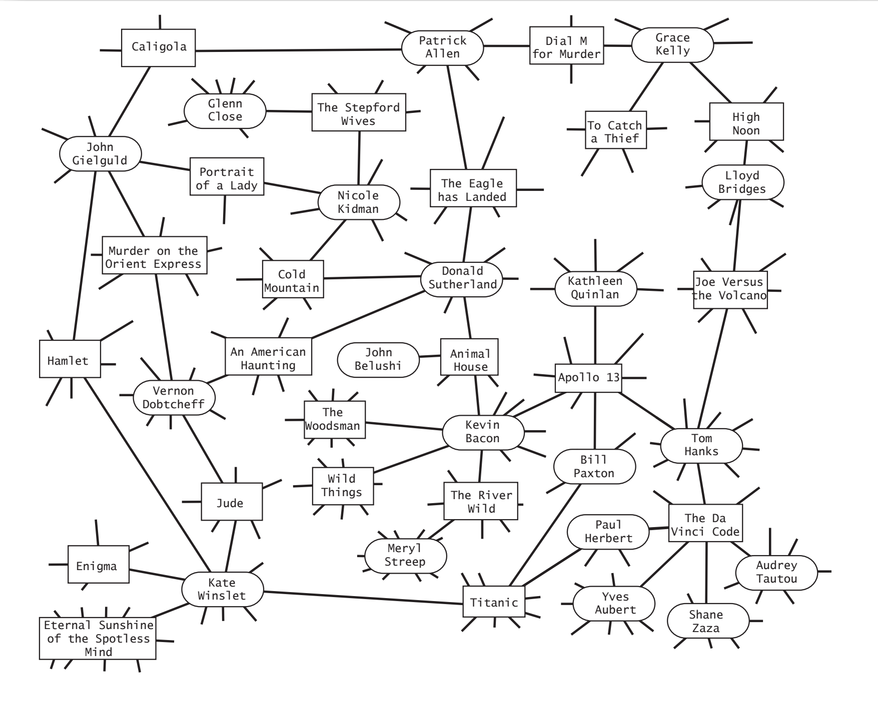
Q. How do we find an *st*-path in a small-world graph?

Applications of small-world graphs

social networks
airlines
roads
neurobiology
evolution
social influence
protein interaction
percolation
internet
electric power grids
political trends
.
.
.

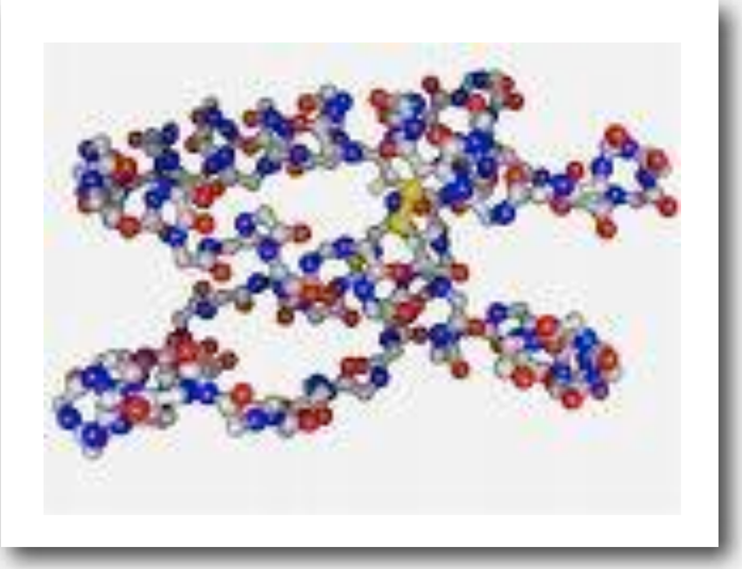
Example 1: Social networks

- infectious diseases
- extensive simulations
- some analytic results
- huge graphs



Example 2: Protein interaction

- small-world model
- natural process
- experimental validation



Finding a path in a small-world graph

is a heavily studied problem

Milgram experiment (1960)

Small-world graph models

- Random (many variants)
- Watts-Strogatz
- Kleinberg

add V random shortcuts to grid graphs and others

A uses ~ log E steps to find a path*

How does 2-way DFS do in this model?

no change at all in graph code just a different graph model

Experiment:

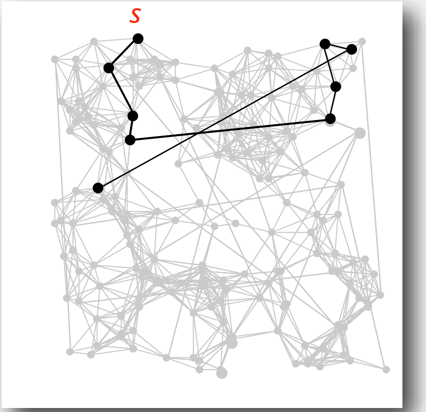
- add M ~ $E^{1/2}$ random edges to an M-by-M grid graph
- use 2-way DFS to find path

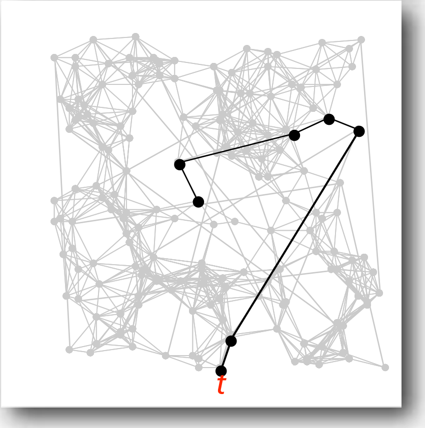
Surprising result: Finds short paths in ~ $E^{1/2}$ steps!

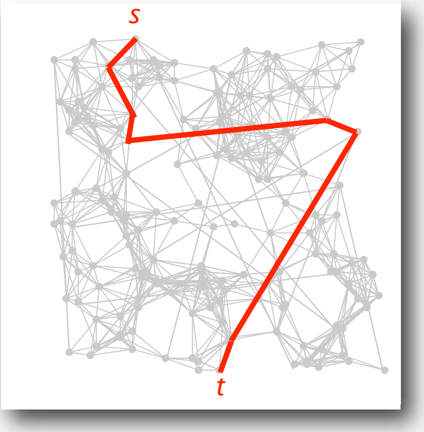
Finding a path in a small-world graph

is much easier than finding a path in a grid graph

Conjecture: Two-way DFS finds a short *st*-path in sublinear time in any small-world graph







Evidence in favor

1. Experiments on many graphs
2. Proof sketch for grid graphs with V shortcuts
   - step 1: 2 $E^{1/2}$ steps ~ 2 $V^{1/2}$ random vertices
   - step 2: like birthday paradox

two sets of 2$V^{1/2}$ randomly chosen vertices are highly unlikely to be disjoint

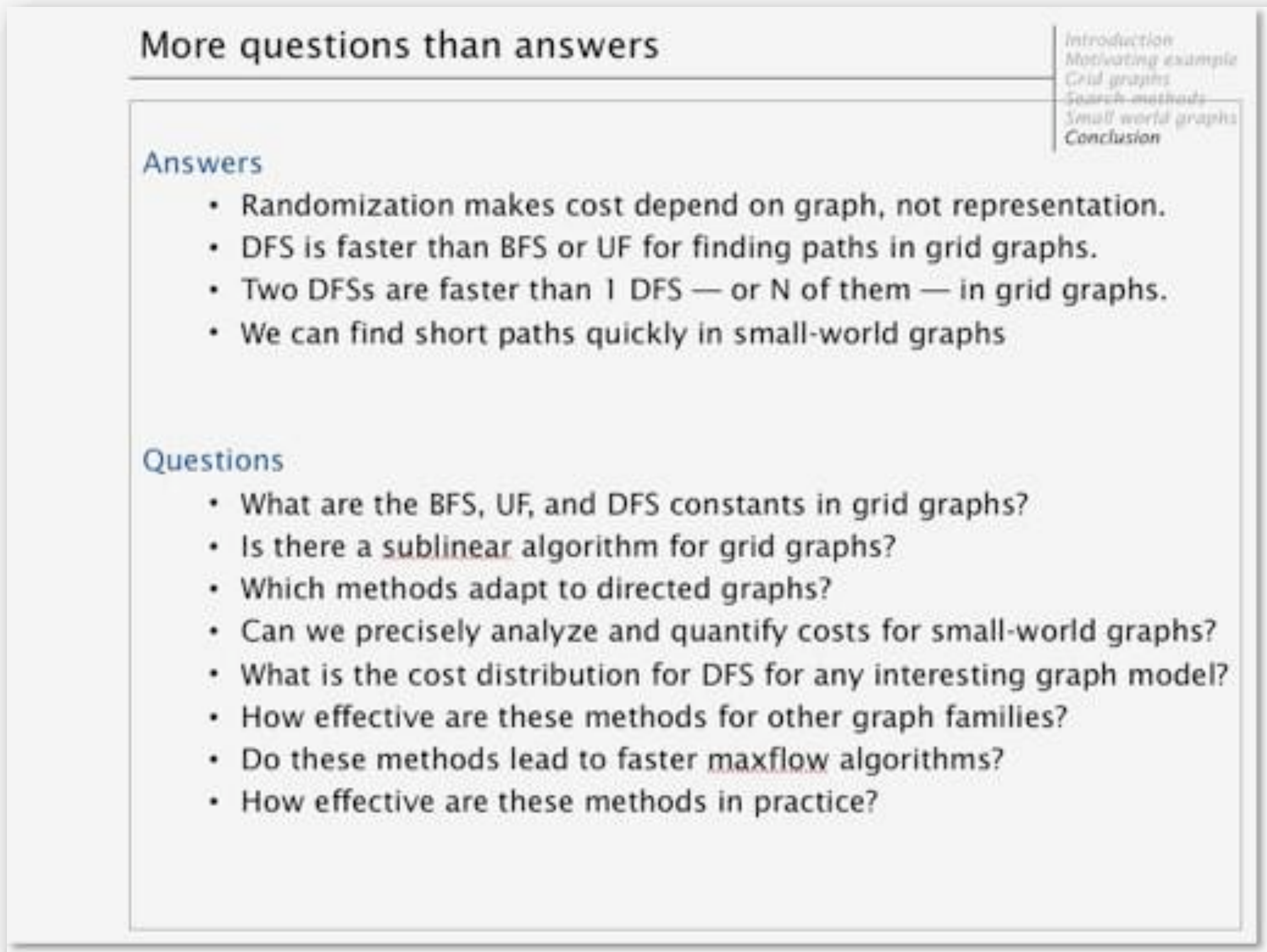
Path length?

Multiple searchers revisited?

Next steps: refine model, more experiments, detailed proofs

Detailed example: paths in graphs

End of “lecture-within-a-lecture”



Lessons

- Data abstraction is for everyone
- We know much less about graph algorithms than you might think
- The scientific method is essential in understanding performance

The role of mathematics

in understanding performance

Worrisome point

- Complicated mathematics seems to be needed for models
- Do all programmers need to know the math?

Good news

- Many people are working on the problem
- Simple universal underlying models are emerging

Appropriate mathematical models

are essential for scientific studies of program behavior

Pioneering work by Don Knuth

Large and active "analysis of algorithms" research community is actively studying models and methods.

Caution: Not all mathematical models are appropriate!

Analytic Combinatorics

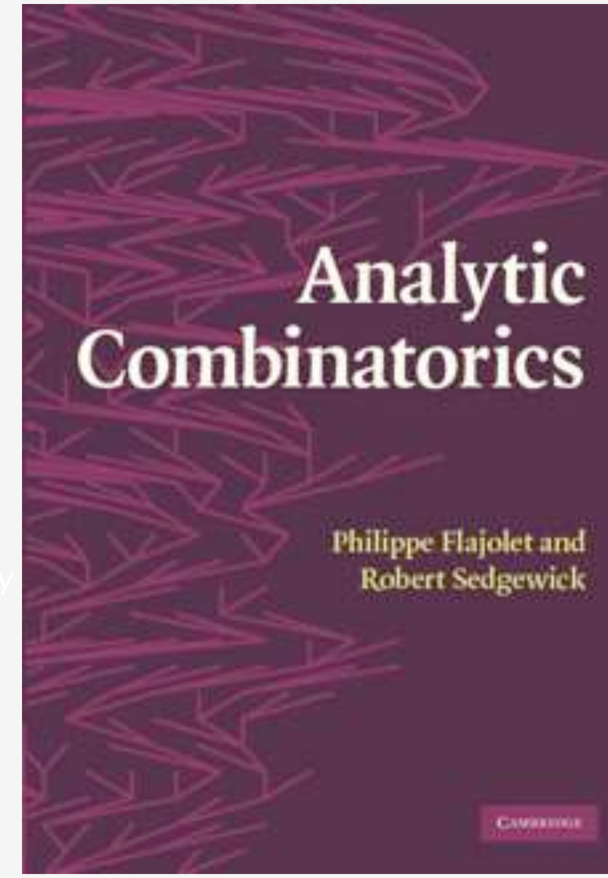
is a modern basis for studying discrete structures

Developed by

Philippe Flajolet and many coauthors (including RS)

based on

classical combinatorics and analysis



Cambridge 2009
also available
on the web

Generating functions (GFs) encapsulate sequences

Symbolic methods treat GFs as formal objects

- formal definition of combinatorial constructions
- direct association with generating functions

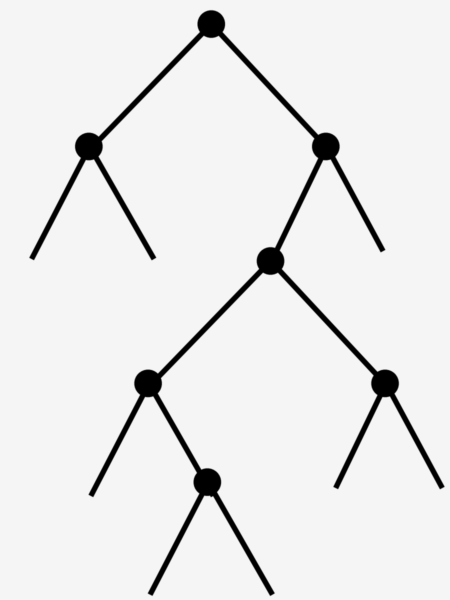
Complex asymptotics treat GFs as functions in the complex plane

- Study them with singularity analysis and other techniques
- Accurately approximate original sequence

Analysis of algorithms: classic example

A binary tree is a node connected to two binary trees.

How many binary trees with N nodes?



Given a recurrence relation

$$B_N = B_0 \, B_{N-1} + \ldots + B_k \, B_{N-1-k} + \ldots + B_{N-1} \, B_0$$

introduce a generating function

$$B(z) \equiv B_0 z^0 + B_1 z^1 + B_2 z^2 + B_3 z^3 + \ldots$$

multiply both sides by z^N and sum to get an equation

$$B(z) = 1 + z\,B(z)^2$$

that we can solve algebraically

$$B(z) = \frac{1 - \sqrt{1 - 4z}}{2z}$$

Quadratic equation

and expand to get coefficients

$$B_N = \frac{1}{N+1}\binom{2N}{N}$$

Binomial theorem

that we can approximate

$$B_N \sim \frac{4^N}{N\sqrt{\pi N}}$$

Stirling's approximation

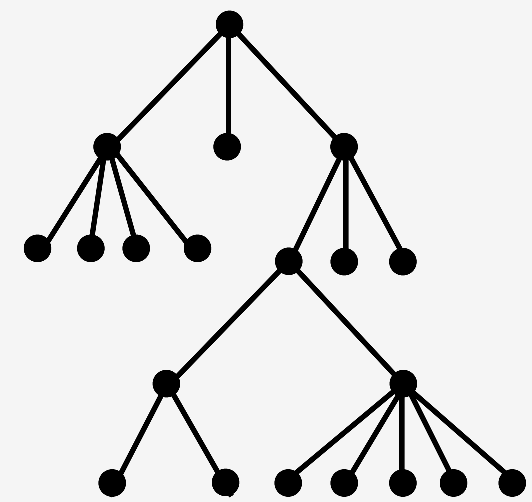
Appears in birthday paradox (and countless other problems) Coincidence?

Basic challenge: need a new derivation for each problem

Analytic combinatorics: classic example

A tree is a node connected to a sequence of trees

How many trees with N nodes?



Combinatorial constructions

$$\langle G\rangle \;=\; \varepsilon \;+\; \langle G\rangle \;+\; \langle G\rangle\times\langle G\rangle \;+\; \langle G\rangle\times\langle G\rangle\times\langle G\rangle \;+\; \ldots$$

directly map to GFs

$$G(z) \;=\; 1 \;+\; G(z) \;+\; G(z)^2 \;+\; G(z)^3 \;+\; \ldots$$

that we can manipulate algebraically

$$G(z) \;=\; \frac{1 - \sqrt{1 - 4z}}{2}$$

by quadratic equation

since $G(z) = \dfrac{1}{1 - G(z)}$,

so $G(z)^2 - G(z) + z = 0$

and treat as a complex function to approximate growth

$$G_N \sim \frac{4^N}{2N\,\Gamma(\tfrac{1}{2})\sqrt{N}} = \frac{4^N}{2N\sqrt{\pi N}}$$

First principle: location of singularity determines exponential growth

Second principle: nature of singularity determines subexponential factor

Analytic combinatorics: singularity analysis

is a key to extracting coefficient asymptotics

Exponential growth factor

- depends on location of dominant singularity
- is easily extracted

Ex: $[z^N](1 - bz)^c = b^N\,[z^N](1 - z)^c$

Polynomial growth factor

- depends on nature of dominant singularity
- can often be computed via contour integration

Ex:

$$[z^N](1 - z)^c = \frac{1}{2\pi i}\int_C \frac{(1 - z)^c}{z^{N+1}}\,dz$$

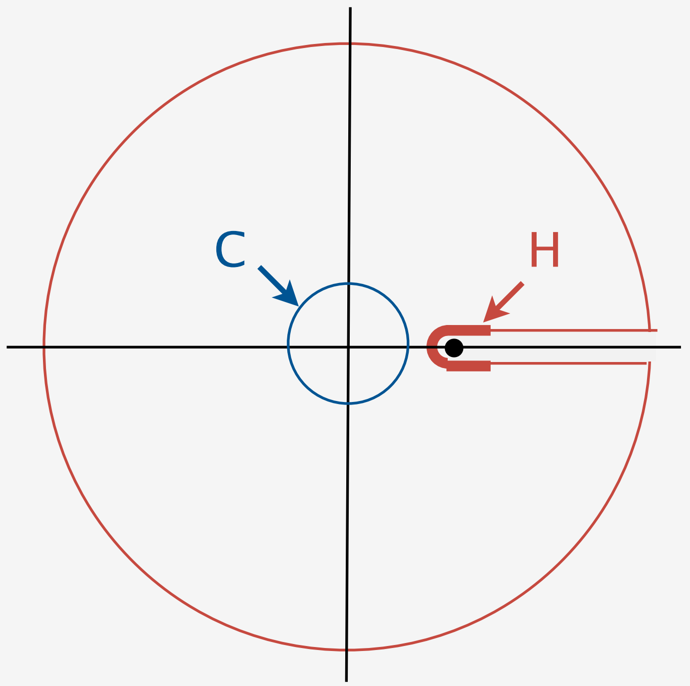
Cauchy coefficient formula

$$\sim \frac{1}{2\pi i}\int_H \frac{(1 - z)^c\,dz}{z^{N+1}}$$

Hankel contour

$$\sim \frac{1}{\Gamma(c)N^{c+1}}$$

many details omitted!



Analytic combinatorics: universal laws

of sweeping generality derive from the same technology

Ex. Context free constructions

Combinatorial constructions

$$\begin{aligned}
\langle G_0 \rangle &= OP_0(\langle G_0 \rangle, \langle G_1 \rangle, \ldots, \langle G_t \rangle) \\
\langle G_1 \rangle &= OP_1(\langle G_0 \rangle, \langle G_1 \rangle, \ldots, \langle G_t \rangle) \\
&\ldots \\
\langle G_t \rangle &= OP_t(\langle G_0 \rangle, \langle G_1 \rangle, \ldots, \langle G_t \rangle)
\end{aligned}$$

like context-free language (or Java data type)

directly map to a system of GFs

$$\begin{aligned}
G_0(z) &= F_0(\, G_0(z),\, G_1(z) \ldots,\ G_t(z)) \\
G_1(z) &= F_1(\, G_0(z),\, G_1(z) \ldots,\ G_t(z)) \\
&\ldots \\
G_t(z) &= F_t(\, G_0(z),\, G_1(z) \ldots,\ G_t(z))
\end{aligned}$$

Groebner-basis elimination

that we can manipulate algebraically to get a single complex function

$$G(z) \equiv G_0(z) = F(\, G_0(z),\ \ldots\ G_t(z)) \sim (1 - z)^{-c}$$

Drmota-Lalley-Woods

that is amenable to singularity analysis

$$G_N \sim a\, b^N N^c$$

for any context-free construction !

Good news: Several such laws have been discovered

Better news: Distributions also available (typically normal, small sigma)

A general hypothesis from analytic combinatorics

The running time of your program is $\sim a \ b^N \ N^c \ (\lg N)^d$

- the constant a depends on both complex functions and properties of machine and implementation
- the exponential growth factor b should be 1
- the exponent c depends on singularities
- the log factor d is reconciled in detailed studies

Why?

- data structures evolve from combinatorial constructions
- universal laws from analytic combinatorics have this form

To compute values:

- $\lg(T(2N)/T(N)) \to c$
- $T(N)/b^N \ N^c \to a$

the doubling test that is the basis for predicting performance!

Plenty of caveats, but provides, in conjunction with the scientific method, a basis for studying program performance

Performance matters in software engineering

Writing a program without understanding performance is like

not knowing where a rocket will go

not knowing the strength of a bridge

not knowing the dosage of a drug

The scientific method is an integral part of software development

Unfortunate facts

Many scientists lack basic knowledge of computer science

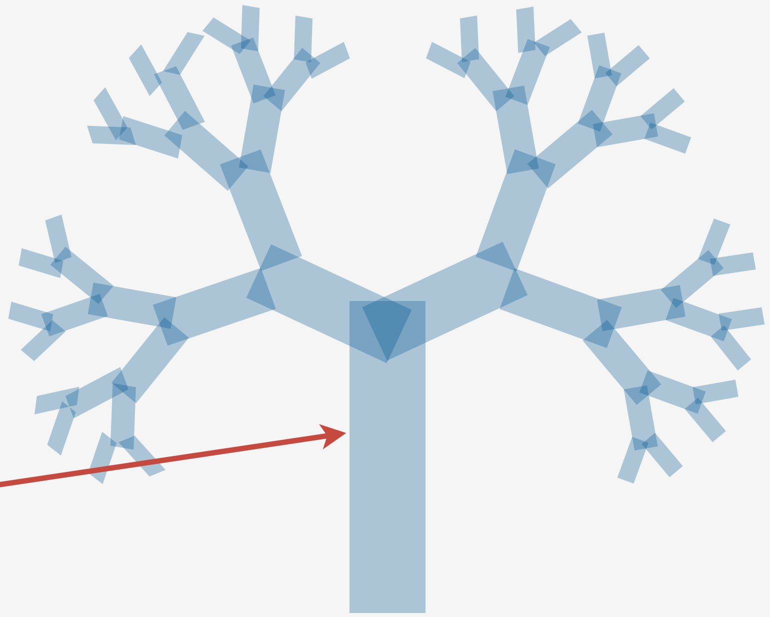
Many computer scientists lack back knowledge of science

1970s: Want to use the computer? Take intro CS.

2000s: Intro CS course relevant only
to future cubicle-dwellers

One way to address the situation

- identify fundamentals
- teach them to all students who need to know them
- as early as possible



Central Thesis (1992)

First-year college students need a computer science course

Computer science embraces a significant body of knowledge that is

- intellectually challenging
- pervasive in modern life
- critical to modern science and engineering

Traditional barriers

- obsolescence
- high equipment costs
- no room in curriculum
- incorrect perceptions about CS
- programming courses bludgeon students with tedium
- one course fits all?
- no textbook

Messages for first-year students

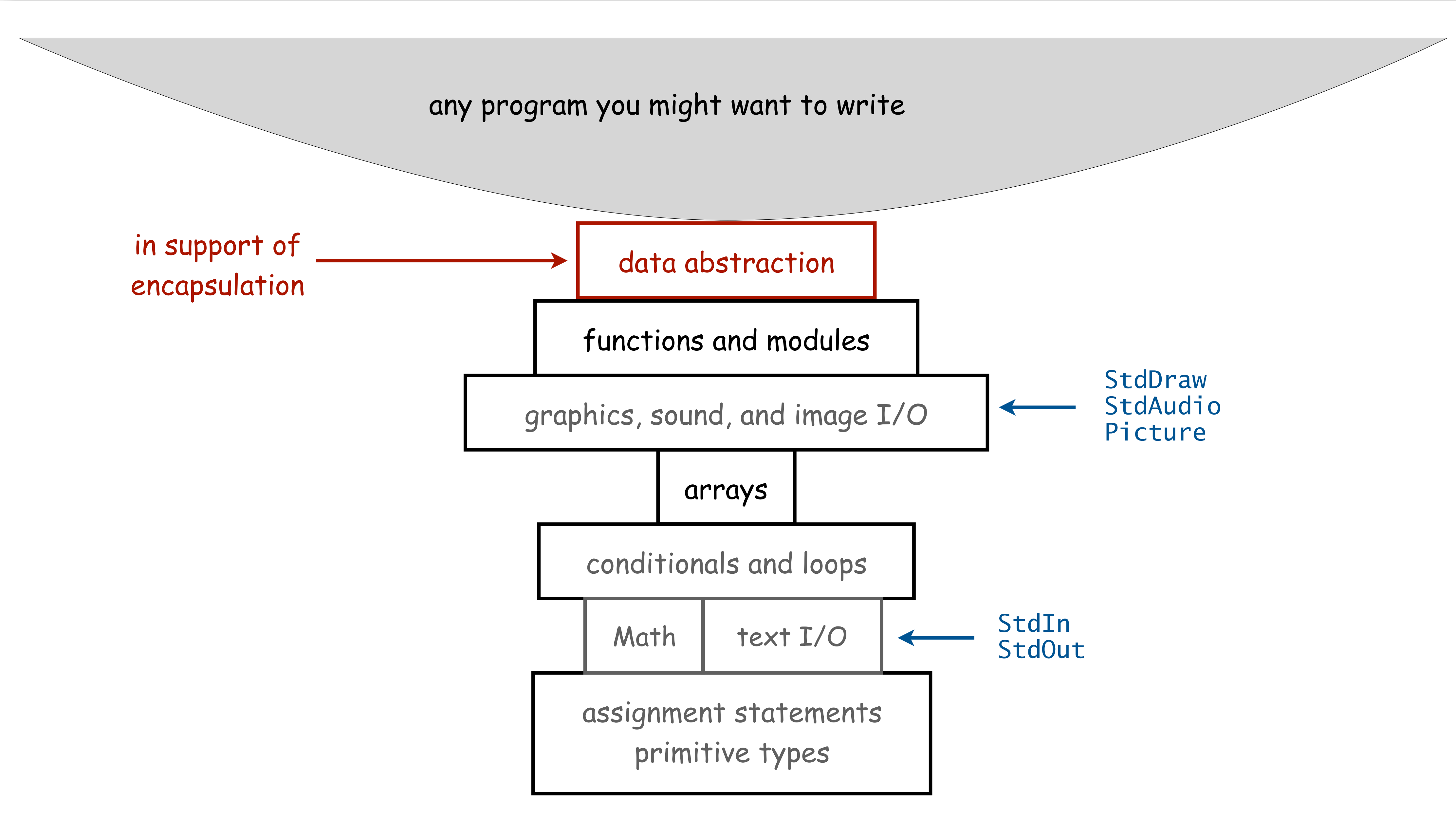
Reading, writing, and computing

Programming is for everyone

- it's easier than most challenges you're facing
- you cannot be successful in any field without it

Computer science is intellectually challenging, worth knowing

Key ingredient: a modern programming model



Basic requirements

- full support of essential components
- freely available, widely used

1990: C/C++, 2010: Java, 2020: ??

CS in scientific context: a few examples

functions	`sin() cos(), log()`
libraries	I/O, data analysis
1D arrays	sound
2D arrays	images
strings	genomes
object-oriented I/O	streaming from the web
OOP	Brownian motion
data structures	small-world phenomenon

Progress report (2010)

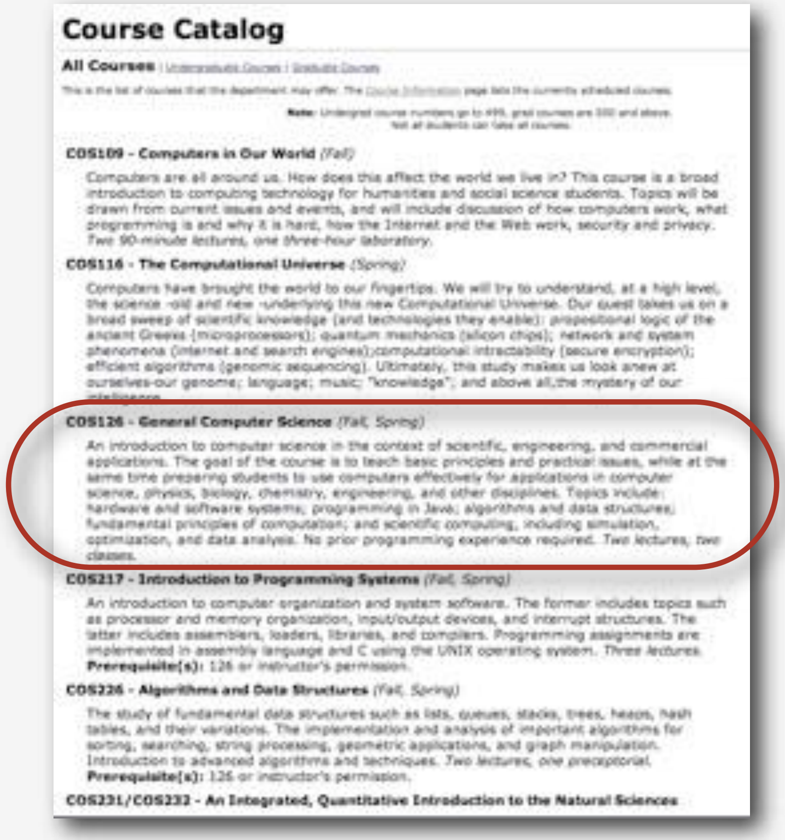
Stable intro CS course for all students

modern programming model

- Basic control structures
- Standard input and output streams
- Drawings, images and sound
- Data abstraction
- Use any computer, and the web

relevant CS concepts

- Understanding of the costs
- Fundamental data types
- Computer architecture
- Computability and Intractability


Course Catalog

All Courses

COS109 - Computers in Our World *(Fall)*

Computers are all around us. How does this affect the world we live in? This course is a broad introduction to computing technology for humanities and social science students. Topics will be drawn from current issues and events, and will include discussion of how computers work, what programming is and why it is hard, how the Internet and the Web work, security and privacy. *Two 90-minute lectures, one three-hour laboratory.*

COS116 - The Computational Universe *(Spring)*

Computers have brought the world to our fingertips. We will try to understand, at a high level, the science -old and new -underlying this new Computational Universe. Our quest takes us on a broad sweep of scientific knowledge (and technologies they enable): propositional logic of the ancient Greeks (microprocessors); quantum mechanics (silicon chips); network and system phenomena (internet and search engines);computational intractability (secure encryption); efficient algorithms (genomic sequencing). Ultimately, this study makes us look anew at ourselves-our genome; language; music; "knowledge"; and above all,the mystery of our intelligence.

COS126 - General Computer Science *(Fall, Spring)*

An introduction to computer science in the context of scientific, engineering, and commercial applications. The goal of the course is to teach basic principles and practical issues, while at the same time preparing students to use computers effectively for applications in computer science, physics, biology, chemistry, engineering, and other disciplines. Topics include: hardware and software systems; programming in Java; algorithms and data structures; fundamental principles of computation; and scientific computing, including simulation, optimization, and data analysis. No prior programming experience required. *Two lectures, two classes.*

COS217 - Introduction to Programming Systems *(Fall, Spring)*

An introduction to computer organization and system software. The former includes topics such as processor and memory organization, input/output devices, and interrupt structures. The latter includes assemblers, loaders, libraries, and compilers. Programming assignments are implemented in assembly language and C using the UNIX operating system. *Three lectures.* **Prerequisite(s):** 126 or instructor's permission.

COS226 - Algorithms and Data Structures *(Fall, Spring)*

The study of fundamental data structures such as lists, queues, stacks, trees, heaps, hash tables, and their variations. The implementation and analysis of important algorithms for sorting, searching, string processing, geometric applications, and graph manipulation. Introduction to advanced algorithms and techniques. *Two lectures, one preceptorial.* **Prerequisite(s):** 126 or instructor's permission.

COS231/COS232 - An Integrated, Quantitative Introduction to the Natural Sciences

scientific content

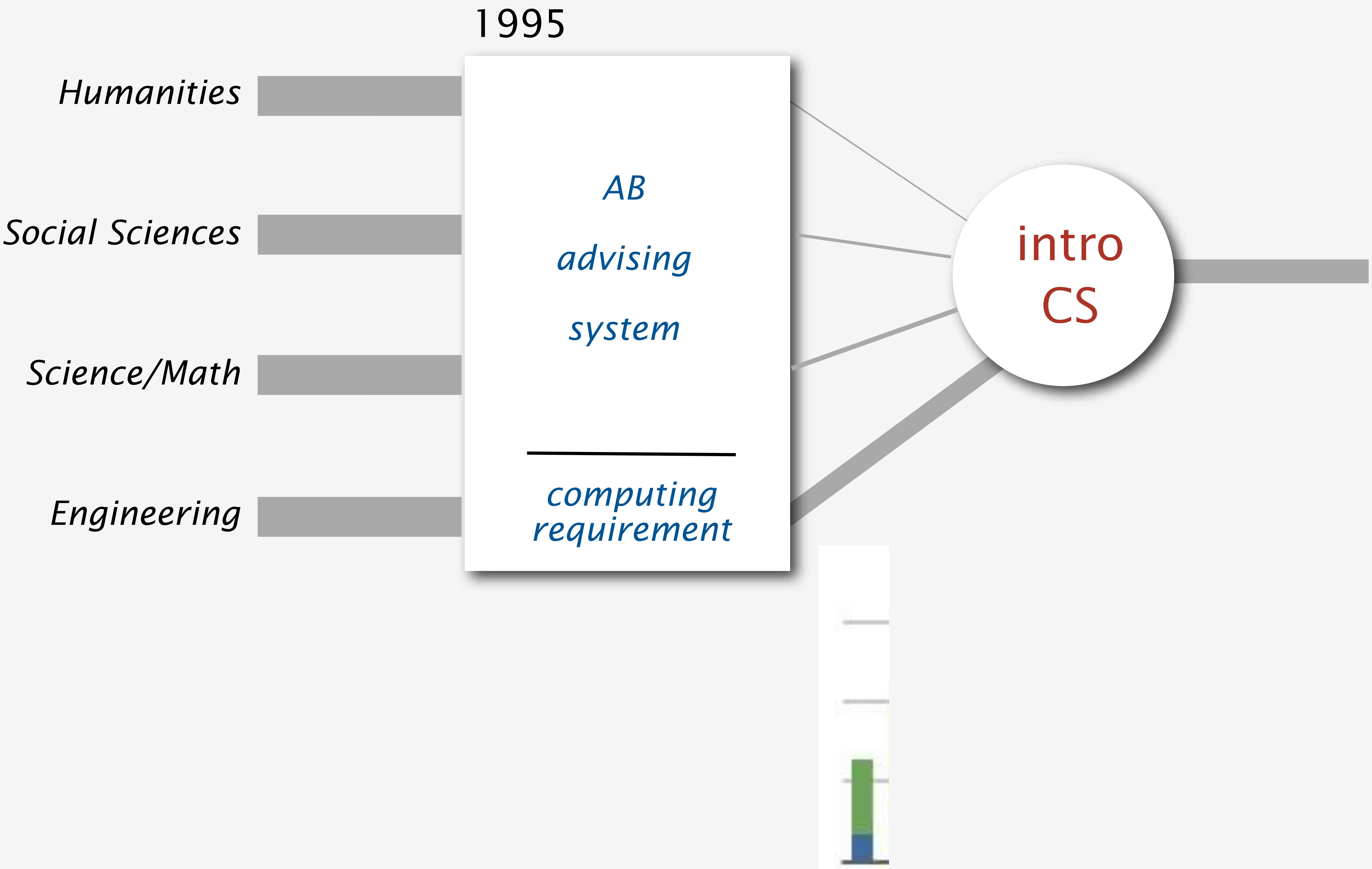
- Scientific method
- Data analysis
- Simulation
- Applications

Goals

- demystify computer systems
- empower students to exploit computation
- build awareness of intellectual underpinnings of CS

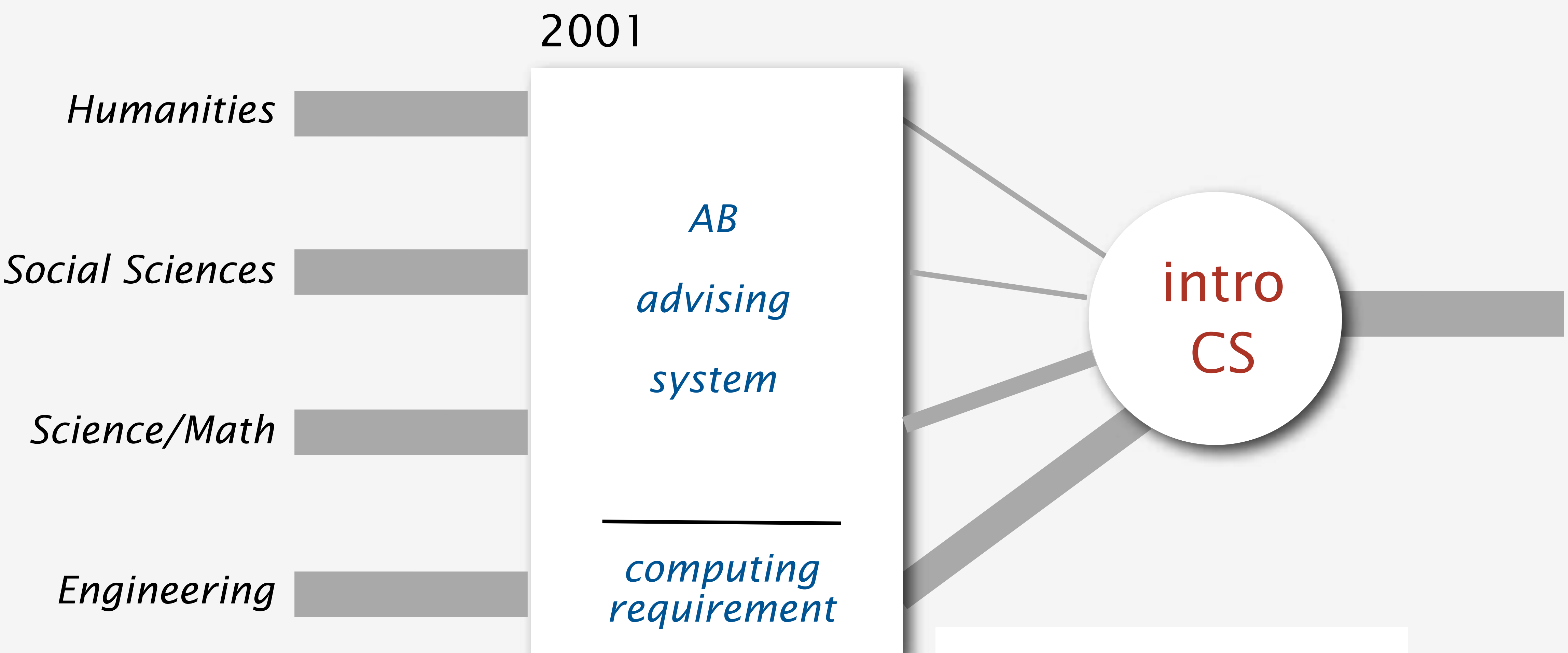
Progress report (continued)

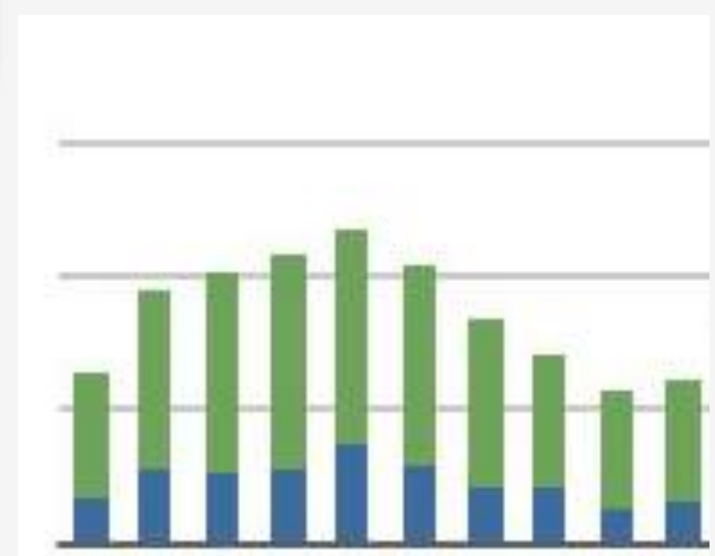
Standard enrollment pattern



Progress report (continued)

Standard enrollment pattern (up and down)

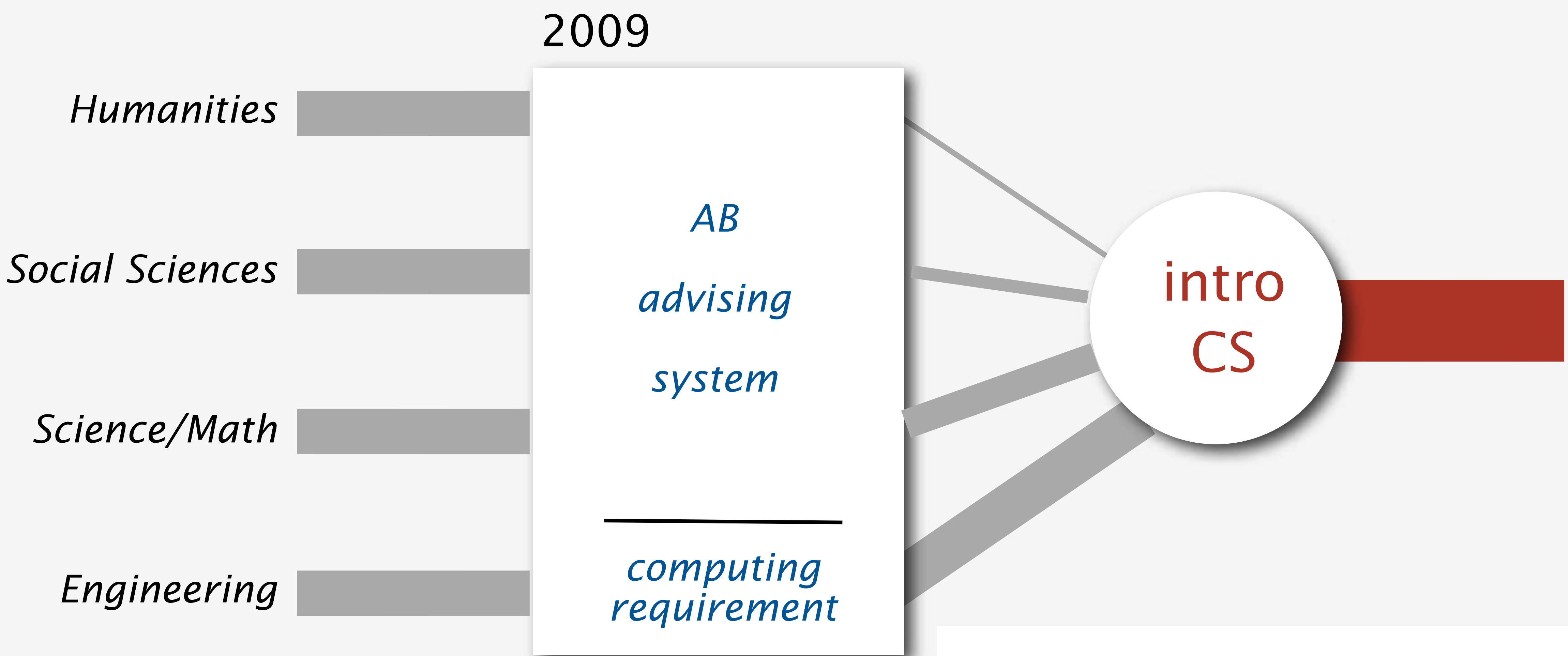




2001

Progress report (continued)

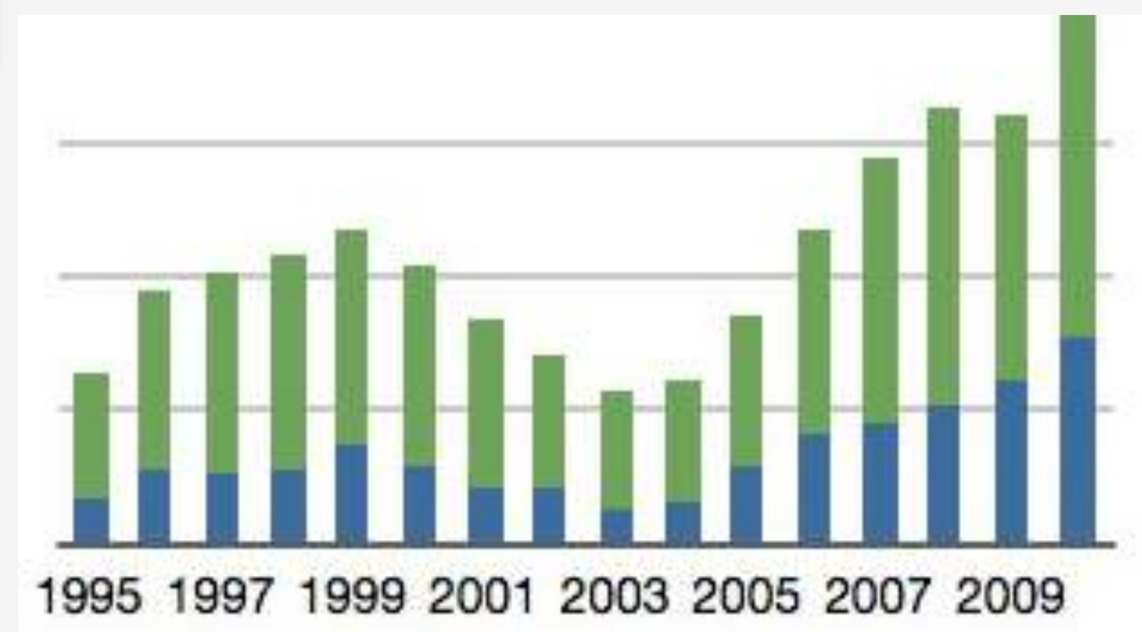
Standard enrollment pattern (up and down), but now is skyrocketing



40% of all Princeton students

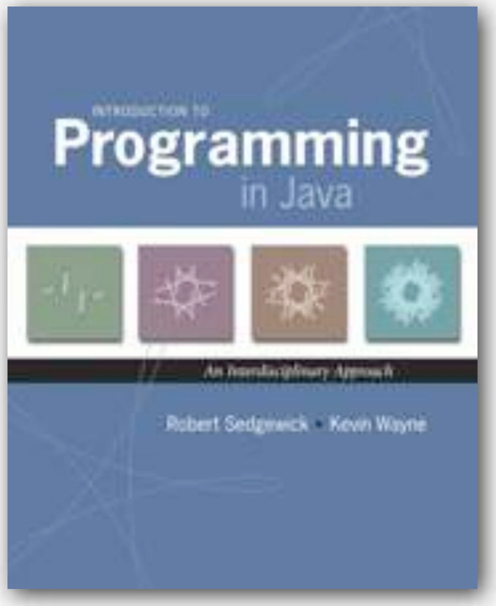
Elective for 40% of enrollees

40% female



Progress report continued (2009)

Textbook and booksite available and widely used



www.cs.princeton.edu/introcs

Anyone can learn the importance of

- modern programming models
- the scientific method in understanding program behavior
- fundamental precepts of computer science
- computation in a broad variety of applications
- preparing for a lifetime of engaging with computation

Textbook

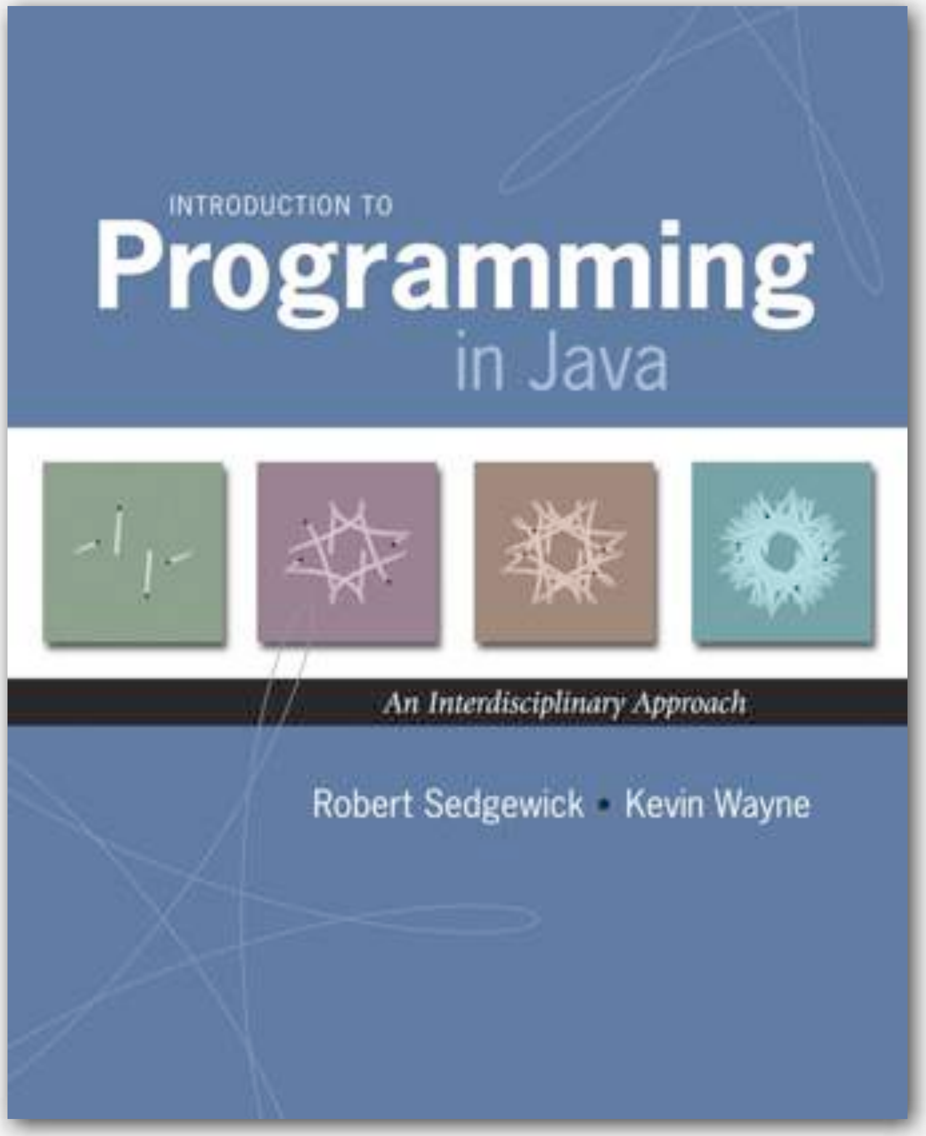
Introduction to Programming in Java:
An interdisciplinary approach
R. Sedgewick and K. Wayne

Elements of Programming
- Your First Program
- Built-in types of Data
- Conditionals and Loops
- Arrays
- Input and Output
- Case Study: Random WebSurfer

Functions and Modules
- Static Methods
- Libraries and Clients
- Recursion
- Case Study: Percolation

Object-Oriented Programming
- Data Types
- Creating DataTypes
- Designing Data Types
- Case Study: Percolation

Algorithms and Data Structures
- Performance
- Sorting and Searching
- Stacks and Queues
- Symbol Tables
- Case Study: Small World



Stay tuned

Introduction to Computer Science

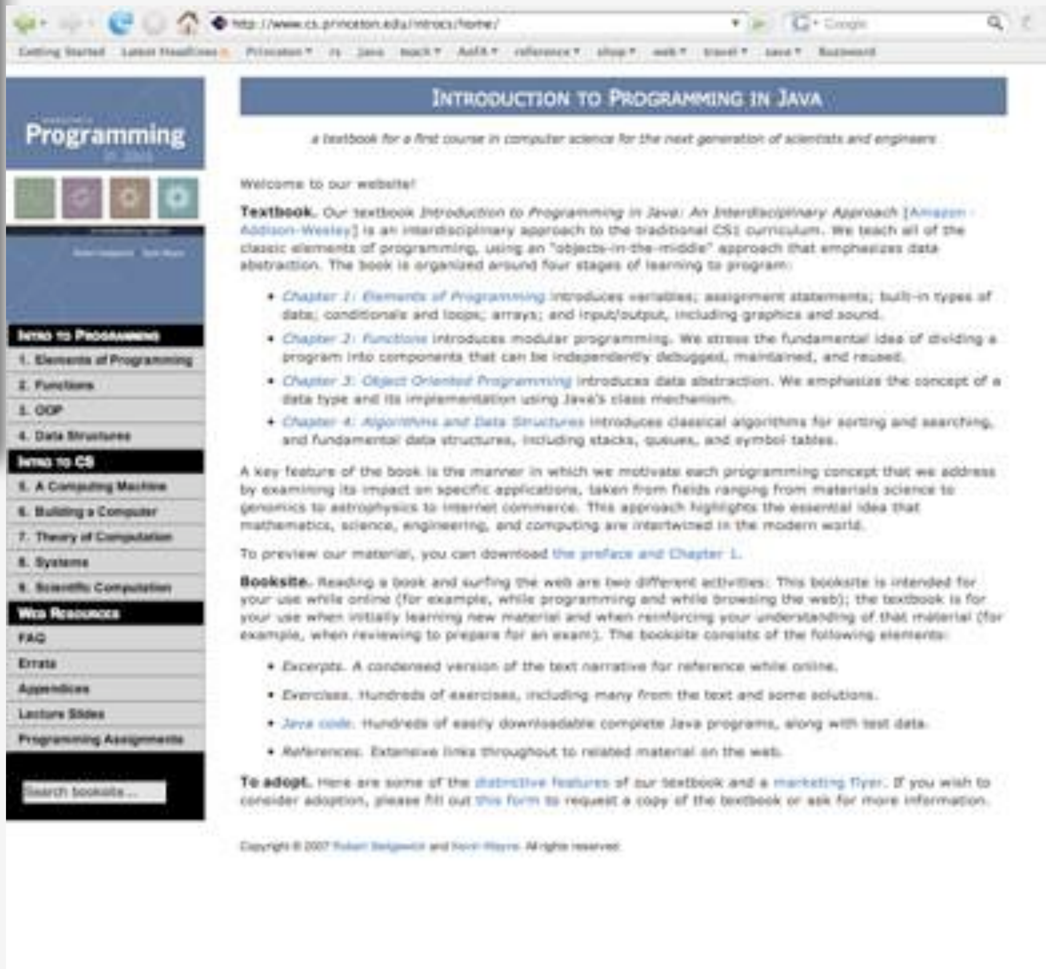
R. Sedgewick and K. Wayne

Booksite

extends text with supplementary material on the web

www.cs.princeton.edu/IntroCS

- Text digests
- Supplementary exercises/answers
- Links to references and sources
- Modularized lecture slides
- Programming assignments
- Demos for lecture and precept
- Simulators for self-study
- Scientific applications



Also: Book development laboratory

- 10000+ files
- 2000+ Java programs
- 50+ animated demos
- 20,000+ files transferred per week

Traditional barriers are falling

Obsolescence?

- focus on concepts reduces language dependencies
- basic features of modern languages are converging

High equipment costs?

- students use their own computers
- basic features of modern OSs are converging

No room in curriculum?

- extensive AP placement makes room
- replace legacy programming courses

Incorrect perceptions about CS?

- yesterday's predictions are today's reality
- young scientists/engineers appreciate importance of CS

Distinctive features of our approach

also address some traditional barriers

No room in curriculum?

- appeal to familiar concepts from HS science and math saves room
- broad coverage provides real choice for students choosing major
- modular organization gives flexibility to adapt to legacy courses
- detailed examples useful throughout curriculum

Incorrect perceptions about CS?

- scientific basis gives students the big picture
- students are enthusiastic about addressing real applications

Excessive focus on programming?

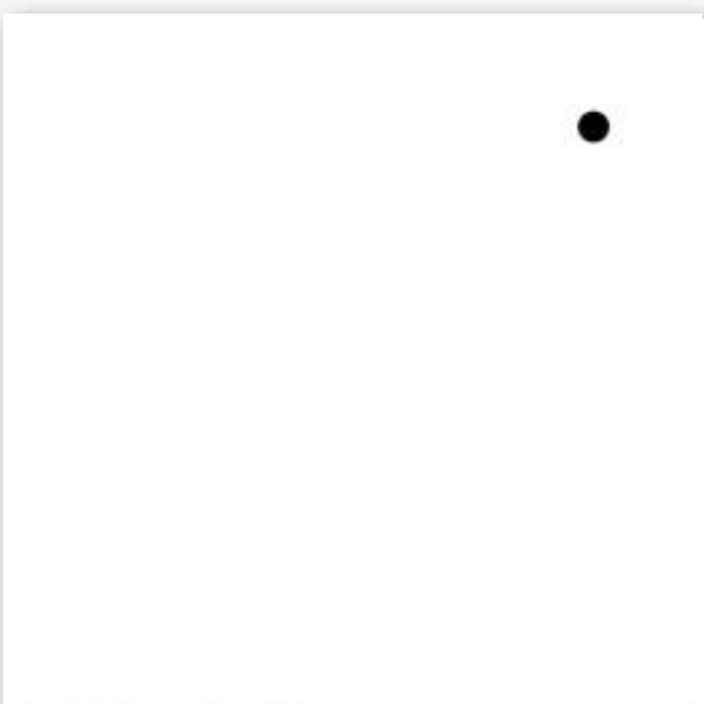
- careful introduction of essential constructs
- nonessential constructs left for later CS courses
- library programming restricted to key abstractions
- taught in context with plenty of other material

Familiar and easy-to-motivate applications

Ideal programming example/assignment

- teaches a basic CS concept
- solves an important problem
- appeals to students' intellectual interest
- illustrates modular programming
- is open-ended

Bouncing ball



Simulation is easy

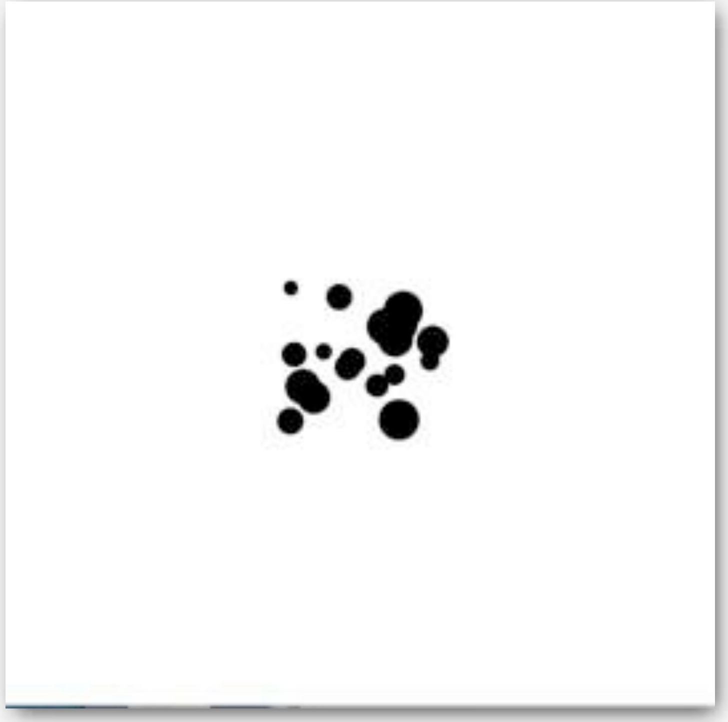
```
public class BouncingBall
{
   public static void main(String[] args)
   {  // Simulate the movement of a bouncing ball.
      StdDraw.setXscale(-1.0, 1.0);
      StdDraw.setYscale(-1.0, 1.0);
      double rx = .480, ry = .860;
      double vx = .015, vy = .023;
      double radius = .05;
      int dt = 20;
      while(true)
      {  // Update ball position and draw it there.
         if (Math.abs(rx + vx) + radius > 1.0) vx = -vx;
         if (Math.abs(ry + vy) + radius > 1.0) vy = -vy;
         rx = rx + vx;
         ry = ry + vy;
         StdDraw.clear();
         StdDraw.filledCircle(rx, ry, radius);
         StdDraw.show(dt);
      }
   }
}
```

Familiar and easy-to-motivate applications

Ideal programming example/assignment

- teaches a basic CS concept
- solves an important problem
- appeals to students' intellectual interest
- illustrates modular programming
- is open-ended

Bouncing balls



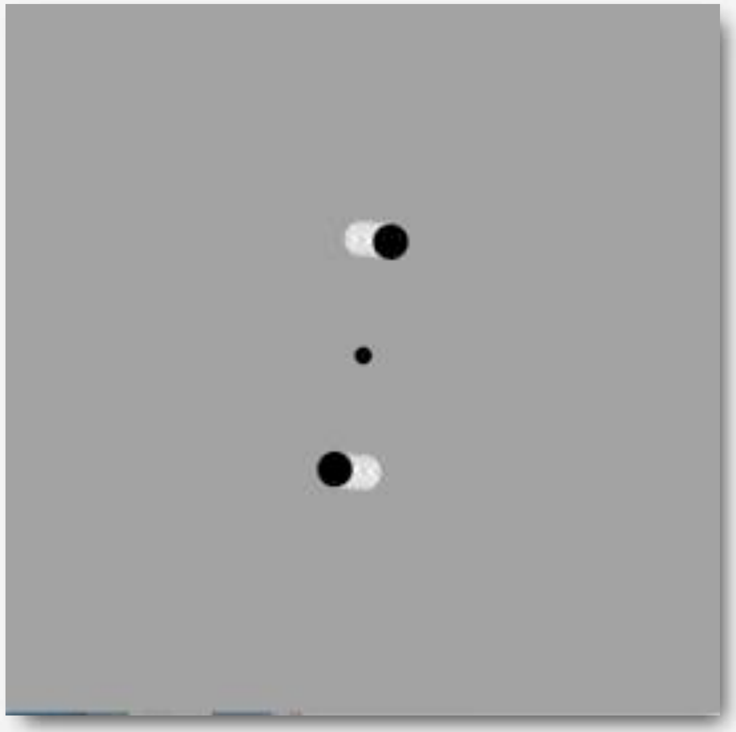
OOP is helpful

Familiar and easy-to-motivate applications

Ideal programming example/assignment

- teaches a basic CS concept
- solves an important problem
- appeals to students' intellectual interest
- illustrates modular programming
- is open-ended

N-body



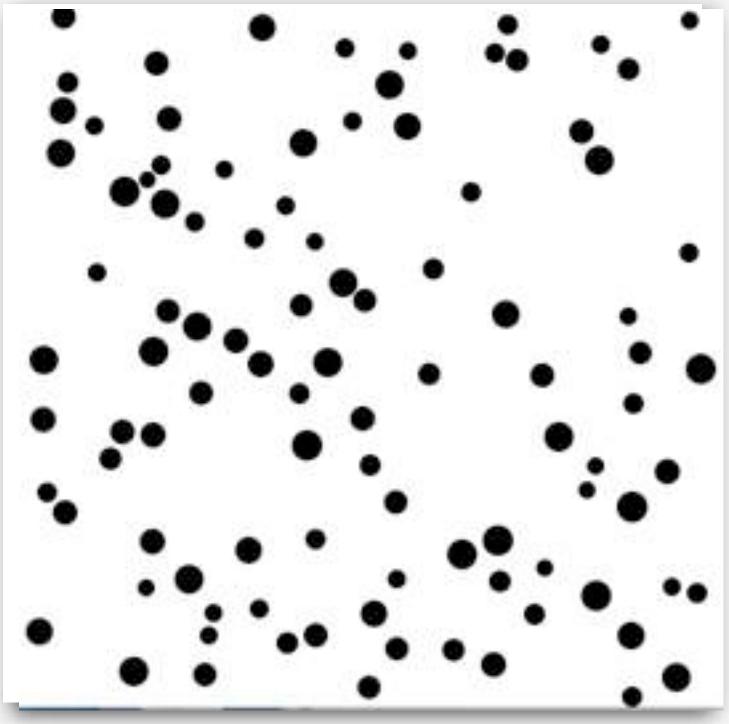
data-driven programs are useful

Familiar and easy-to-motivate applications

Ideal programming example/assignment

- teaches a basic CS concept
- solves an important problem
- appeals to students' intellectual interest
- illustrates modular programming
- is open-ended

Bose-Einstein colliding particle siimulation



*a poster child for **priority queue** abstraction*

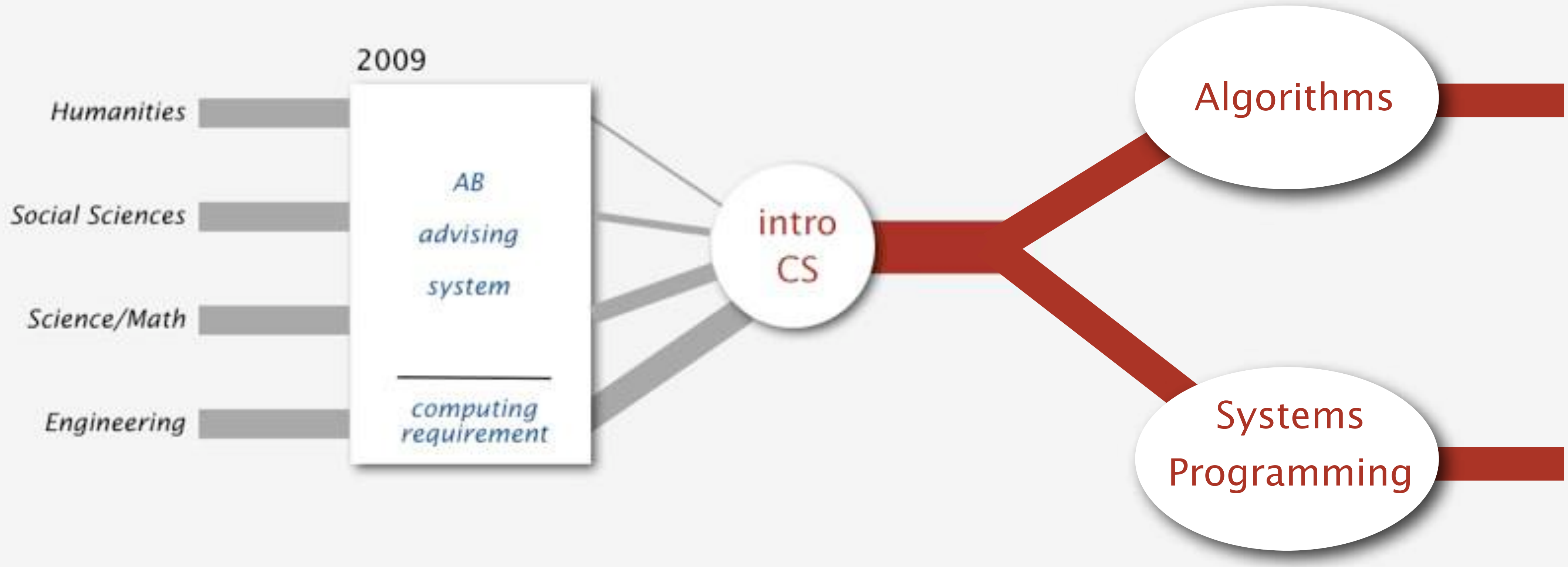
efficient algorithms are necessary

Scientific method is not harmful

"Algorithms" and "Systems Programming" benefit from the approach.

About half of the IntroCS students take both!

Half of those pursue a certificate program in Applications in Computing



Summary

Computer science embraces a significant body of knowledge that is pervasive in modern life and critical to every students' education

Embracing, supporting, and leveraging science in a single intro CS course can serve large numbers of students.

Proof of concept: Intro CS at Princeton

- 40% of Princeton students in a single intro course
- Stable content for a decade

Next goal: 40% of US college students

- Classical textbook model
- New media
- Evangelization
- Viral spread of content

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Why not C/C++ ?

A. Low-level pro tools; pros can learn later.

A. Been there, done that.

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Why not C/C++ ?

A. Low-level pro tools; pros can learn later.

A. Been there, done that.

Q. Why not Python?

A. Poor data abstraction; everyone needs layers of abstraction.

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Why not C/C++ ?

A. Low-level pro tools; pros can learn later.

A. Been there, done that.

Q. Why not Python?

A. Poor data abstraction; everyone needs layers of abstraction.

Q. Why not Matlab?

A. Not free.

A. Poor data abstraction ("i = 0").

A. Not so relevant to students who do not know linear algebra.

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Our students are not as smart as Princeton students.

A. They're all relatively smart teenagers.

A. You use the same calculus and physics texts that we use.

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Our students are not as smart as Princeton students.

A. They're all relatively smart teenagers.

A. You use the same calculus and physics texts that we use.

Q. The math and science is too difficult/advanced.

A. We are just leveraging high-school math and science.

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Our students are not as smart as Princeton students.

A. They're all relatively smart teenagers.

A. You use the same calculus and physics texts that we use.

Q. The math and science is too difficult/advanced.

A. We are just leveraging high-school math and science.

Q. How do students learn to program, if you give them the code?

A. They see numerous examples.

A. They face a new, interesting challenge each week (see booksite).

FAQs

Q. Why Java?

A. Widely available, easily installed on any machine.

A. Modern language, widely used in real applications.

Q. Our students are not as smart as Princeton students.

A. They're all relatively smart teenagers.

A. You use the same calculus and physics texts that we use.

Q. The math and science is too difficult/advanced.

A. We are just leveraging high-school math and science.

Q. How do students learn to program, if you give them the code?

A. They see numerous examples.

A. They face a new, interesting challenge each week (see booksite).

Q. How do I use the booksite?

A. 17-year olds have absolutely no trouble doing it!