Algorithms for the masses

Robert Sedgewick
Princeton University
This talk is dedicated to the memory of Philippe Flajolet

Philippe Flajolet 1948–2011
Prelude: my first job

1969 Western Electric Engineering Research Center

PDP-9
- 18-bit words
- 16K memory
- switches+lights
- paper tape reader/punch
- display w/ lightpen

Task: Drafting (Illustration)

Workflow
- write cross-assembler in IBM 360/50 assembly
- write Illustrator application in PDP-9 assembly
- load paper tape into PDP-9
- run application
Prelude: my first job

Problem: Cross-assembler too slow
Solution: Binary search!
  • M searches, N symbols M >> N
  • Improved running time from ~MN to ~MlgN

Lesson 1: Good algorithms matter
Lesson 2: Not many programmers appreciate that fact
Brief history of Algorithms
Algorithms (central thesis)

1975: What are the algorithms that everyone should know?

“Everyone” means “everyone”
  • scientists
  • engineers
  • mathematicians
  • software/hardware designers
  • cryptanalysts
  • COBOL programmers
  • ...

Context
  • IBM 360/50
  • Algol W
  • one run per day
<table>
<thead>
<tr>
<th>Year</th>
<th>Edition</th>
<th>Code Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1982</td>
<td>1st</td>
<td>compiles</td>
</tr>
<tr>
<td>1986</td>
<td>2nd</td>
<td>runs</td>
</tr>
<tr>
<td>1994</td>
<td>3rd</td>
<td>performance comparisons within reasonable practical model</td>
</tr>
<tr>
<td>2011</td>
<td>4th</td>
<td>[this talk]</td>
</tr>
</tbody>
</table>
Four challenges

I. Many algorithms implemented/tested in back rooms, not open literature

II. Need appropriate mathematical models

III. Masses can’t program, don’t know CS

IV. How to disseminate?
I. Scientific method
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

Important lessons of the past several decades
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
- engineers
- scientists
- programmers

- indexing and search
- Bose-Einstein model
- N-body
- signal processing
- string matching for genomics
- natural language analysis
[ very long list ]
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. 2010s: Uses scientific method?

Algorithm designer who does not experiment gets lost in abstraction
Software developer who ignores cost risks catastrophic consequences
Scientist/engineer needs to control costs of experiments/simulations
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 for a reasonable model

\[
\begin{array}{ccc}
V & E & C \\
\text{vertices} & \text{edges} & \text{max capacity}
\end{array}
\]

How many augmenting paths?

- shortest: \( \frac{VE}{2} \)
- max capacity: \( 2E \log C \)

How many steps to find each path?

\( E \) (upper bound)
Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters for a reasonable model

V = 177  E = 2000  C = 100

upper bound  for example

| shortest     | VE/2     | 177,000  |
| VC           |         | 17,700   |
| max capacity | 2E lg C  | 26,575   |

How many steps to find each path?

2000 (upper bound)
Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters for a reasonable model

\[ V = 177 \quad E = 2000 \quad C = 100 \]

<table>
<thead>
<tr>
<th>vertices</th>
<th>edges</th>
<th>max capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>177</td>
<td>2000</td>
<td>100</td>
</tr>
</tbody>
</table>

How many augmenting paths?

<table>
<thead>
<tr>
<th>upper bound</th>
<th>for example</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortest</td>
<td>VE/2</td>
<td>177,000</td>
</tr>
<tr>
<td>VC</td>
<td>17,700</td>
<td></td>
</tr>
<tr>
<td>max capacity</td>
<td>2E \lg C</td>
<td>26,575</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

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

Prediction of total cost is a factor of \textbf{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 for a reasonable model

\[
\begin{array}{ccc}
V & E & C \\
\text{vertices} & \text{edges} & \text{max capacity}
\end{array}
\]

How many augmenting paths?

\[
\begin{align*}
\text{shortest} & : VE/2 \\
\text{max capacity} & : 2E \lg C
\end{align*}
\]

How many steps to find each path? \(E\) (upper bound)

Warning: Such analyses are useless for predicting performance or comparing algorithms.
Motivating example: lessons

Goals of algorithm analysis

- predict performance (running time) or
- 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!
O-notation considered harmful

How to predict performance (and to compare algorithms)?

**Not** the scientific method: \( O \)-notation

**Theorem**: Running time is \( O(N^c) \)

- not at all useful for predicting performance

**Scientific method** calls for tilde-notation.

**Hypothesis**: Running time is \( \sim a N^c \)

- an effective path to predicting performance (stay tuned)

\( O \)-notation is useful for many reasons, BUT

**Common error**: Thinking that \( O \)-notation is useful for predicting performance.
Surely, we can do better

A typical exchange in Q&A

RS (in a talk): O-notation considered harmful. Cannot use it to predict performance.

Q: ?? O(N log N) surely beats O(N²)

RS: Not by the definition. O expresses upper bound.

Q: So, use Theta.

RS: Still (typically) bounding the worst case. Is the input a worst case?

Q: (whispers to colleague) I’d use the Θ(N log N) algorithm, wouldn’t you?
Galactic algorithms

R.J. Lipton: A galactic algorithm is one that will never by used in practice

Why? Any effect would never be noticed in this galaxy

Ex. Chazelle’s linear-time triangulation algorithm
  • theoretical tour-de-force
  • too complicated to implement
  • cost of implementing would exceed savings in this galaxy, anyway

One blogger’s conservative estimate: 75% SODA, 95% STOC/FOCS are galactic

OK for basic research to drive agenda, BUT

**Common error:** Thinking that a galactic algorithm is useful in practice.
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 Algorithm B. I don’t care about Google.
II. Analytic Combinatorics
Analysis of algorithms and analytic combinatorics

Appropriate mathematical models are essential for scientific studies of program behavior

Pioneering work by Don Knuth

Active AoA community is building on classical research in

- probability
- combinatorics
- analysis
- information theory

and is developing new models, methods, and applications
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

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?

Develop a recurrence relation.

\[
B_N = \sum_{0 \leq k < N} B_k B_{N-1-k} \quad \text{with} \quad B_0 = 0
\]

Then introduce a generating function.

\[
B(z) = \sum_{k \geq 0} z^k
\]

Multiply both sides by \( z^N \) and sum to get an equation that we can solve algebraically.

\[
B(z) = 1 + zB(z)^2
\]

Quadratic equation

and expand to get coefficients that we can approximate.

\[
B_N = \frac{1}{N+1} \binom{2N}{N}
\]

Binomial theorem

\[
B_N \sim \frac{4^N}{N \sqrt{\pi N}}
\]

Stirling’s approximation

Challenge: Efficiently teach basic math skills behind such derivations.
Analytic Combinatorics: classic example

A **binary tree** is a node connected to two binary trees. How many binary trees with N nodes?

Develop a **combinatorial construction**, which *directly maps* to a **GF equation** that we can **manipulate algebraically** and treat as a function in the complex plane **directly approximate** via **singularity analysis**.

**Challenge**: Develop an effective calculus for such derivations.

$$< B >= \epsilon + < B > \times \bullet \times < B >$$

$$B(z) = 1 + zB(z)^2$$

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

$$B_N = \frac{4^N}{N \Gamma(1/2) \sqrt{N}} \sim \frac{4^N}{N \sqrt{\pi N}}$$
Complexification

Assigning complex values to the variable $z$ in a GF gives a method of analysis to estimate the coefficients.

The singularities of the function determine the method.

<table>
<thead>
<tr>
<th>singularity type</th>
<th>method of analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>meromorphic (just poles)</td>
<td>Cauchy (elementary)</td>
</tr>
<tr>
<td>fractional powers</td>
<td>Cauchy (Flajolet-Odlyzko)</td>
</tr>
<tr>
<td>logarithmic</td>
<td>saddle point</td>
</tr>
<tr>
<td>none (entire function)</td>
<td></td>
</tr>
</tbody>
</table>

First Principle. Exponential growth of a function’s coefficients is determined by the location of its singularities.

Second Principle. Subexponential factor in a function’s coefficients is determined by the nature of its singularities.
Q. Wait, didn’t you say that the masses don’t need to know all that math?

RS. Well, there is one thing…
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

Plenty of caveats, but provides, in conjunction with the scientific method, a basis for studying program performance
Computing the constants (the hard way)

Knuth showed that it is possible *in principle* to precisely predict running time

- develop a mathematical model for the frequency of execution of each instruction in the program
- determine the time required to execute each instruction
- multiply and sum

\[ T(N) \sim aN^c \]

**Hypothesis**

- **engineer’s part of the constant**
  - harder to determine now than in the 1970s

- **mathematician’s part of the constant**
  - easier to determine now than in the 1970s
Computing the constants (the easy way)

Run the program!

**Hypothesis:** $T(N) \sim a N^c$  

*Note: log factors are more difficult*

1. Implement the program

2. Compute $T(N_0)$ and $T(2N_0)$ by running it

3. Calculate $c$ as follows:

$$\frac{T(2N_0)}{T(N_0)} \sim \frac{a(2N_0)^c}{aN_0^c} = 2^c$$

\[ \lg(T(2N_0)/T(N_0)) \to c \] as $N_0$ grows

4. Calculate $a$ as follows:

$$T(N_0)/N_0^c \to a \] as $N_0$ grows
Predicting performance (the easy way)

Don’t bother computing the constants!

Hypothesis: $T(N) \sim a N^c$

1. Implement the program

2. Run it for $N_0$, $2N_0$, $4N_0$, $8N_0$, ...

3. Note that ratio of running times approaches $2^c$

$$\frac{T(2N_0)}{T(N_0)} \sim \frac{a(2N_0)^c}{aN_0^c}$$

$$= 2^c$$

4. Multiply by $2^c$ to predict next value

Plenty of caveats, but provides a basis for teaching the masses about program performance

---

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1.1 sec</td>
</tr>
<tr>
<td>2000</td>
<td>4.5 sec</td>
</tr>
<tr>
<td>4000</td>
<td>18 sec</td>
</tr>
<tr>
<td>8000</td>
<td>73 sec</td>
</tr>
<tr>
<td>16000</td>
<td>295 sec</td>
</tr>
</tbody>
</table>

if log factors exist, estimate improves as $N$ grows

borders on malpractice *not* to do so!
III. Introduction to CS
The masses

Scientists, engineers and modern programmers need

- extensive specialized knowledge in their field
- an understanding of the scientific method.

They also need to know how to

- write programs
- design and analyze algorithms

Do they need to know?

- Detailed analysis
- Galactic algorithms
- Overly simple input models

They do need to know

- Classic algorithms
- Realistic input models and randomization
- How to predict performance and compare algorithms
Unfortunate facts

Many scientists/engineers lack basic knowledge of computer science
Many computer scientists lack back knowledge of science/engineering

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

1990s: 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
Intro course model: typical

- CS for CS majors
- CS for physicists
- CS for math majors
- CS for biochemists
- CS for economists
- CS for rocket scientists
- CS for idiots
- CS for poets
- CS for EE
- CS for civil engineers
Intro course model: RS view

CS for everyone

Original motivation (1992)
Why not?
Works for biology, math, physics, economics.
Responsibility to identify and teach fundamental tenets of discipline.
Current status (2012)

[Joint work with Kevin Wayne since 2000]

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 and booksite available and widely used [stay tuned]
Messages for first-year students

**Reading, writing, and computing**

Programming is for everyone, including you
- it is easier than most challenges you’re facing
- you cannot be successful in any field without it

**Performance matters**

There is more to computer science than programming

**Computer science is intellectually challenging, worth knowing**
Key ingredient: a modern programming model

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

**CS in scientific context**

Ideal programming example/assignment
- teaches a basic CS concept
- solves an important problem
- intellectually engaging and appealing
- illustrates modular programming
- is open-ended

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>functions</td>
<td><code>sqrt()</code>, <code>log()</code></td>
</tr>
<tr>
<td>libraries</td>
<td><code>I/O</code>, <code>data analysis</code></td>
</tr>
<tr>
<td>1D arrays</td>
<td><code>sound</code></td>
</tr>
<tr>
<td>2D arrays</td>
<td><code>images</code></td>
</tr>
<tr>
<td>recursion</td>
<td><code>fractal models</code></td>
</tr>
<tr>
<td>strings</td>
<td><code>genomes</code></td>
</tr>
<tr>
<td>I/O streams</td>
<td><code>web resources</code></td>
</tr>
<tr>
<td>OOP</td>
<td><code>Brownian motion</code></td>
</tr>
<tr>
<td>data structures</td>
<td><code>small-world</code></td>
</tr>
</tbody>
</table>
Familiar and easy-to-motivate applications

Ideal programming example/assignment

• teaches a basic CS concept
• solves an important problem
• intellectually engaging and appealing
• illustrates modular programming
• is open-ended

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
Distinctive features of our approach

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 abstractns
  • taught in context with plenty of other material
Distinctive features of our approach

address some traditional barriers

One course fits all?

• few students get adequate CS in high school nowadays
• 90+ percent on level playing field by midterms
• open-ended assignments appeal even to experienced programmers
• not harmful for CS students to learn scientific context before diving into abstraction

CS is for cubicle-dwellers?

• “learned more in this course than in any other”
• “came here to study physics/math/bio/econ, now I want to do CS”
• “cool”
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

**Goals**

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

**scientific content**

- Scientific method
- Data analysis
- Simulation
- Applications
Progress report

2008: Enrollments are up. Is this another “bubble”?

COS 126 enrollments
Progress report

2009: Maybe.
Progress report

2012: Enrollments are skyrocketing.

COS 126 enrollments

Enrollments now are twice what they were at the height of the bubble.

Seemed like a lot at the time.
Who are they?

Over half of all Princeton students.

**PROGRAMMING EXPERIENCE**
- None: 35%
- Some: 62%
- Lots: 3%

**CLASS**
- First-year: 70%
- Sophomore: 18%
- Junior: 4%
- Senior: 4%

**INTENDED MAJOR**
- Other Science/Math: 37%
- Other Engineering: 24%
- Humanities: 18%
- Social sciences: 18%
- CS: 11%
IV. Future of publishing
Seismic changes are afoot

Books?

Libraries?

Textbooks?

Why try to write a new textbook in this environment?
. . .Sit in your local coffee shop, and your laptop can tell you a lot. If you want deeper, more local knowledge, you will have to take the narrower path that leads between the lions and up the stairs. There—as in great libraries around the world—you’ll use all the new sources, the library’s and those it buys from others, all the time. You’ll check musicians’ names and dates at Grove Music Online, read Marlowe’s “Doctor Faustus” on Early English Books Online, or decipher Civil War documents on Valley of the Shadow. But these streams of data, rich as they are, will illuminate, rather than eliminate, books and prints and manuscripts that only the library can put in front of you. The narrow path still leads, as it must, to crowded public rooms where the sunlight gleams on varnished tables, and knowledge is embodied in millions of dusty, crumbling, smelly, irreplaceable documents and books.
RS: Think about the future

The New Yorker
Letter to the editor

While Grafton’s reservations about putting knowledge online are well taken, I would also point out that there is quite a bit going on now in the academic world that doesn’t have much to do with old books. Indeed, as the author of many books, I wonder whether perhaps the book is not quite sacred as a means of disseminating knowledge. What is the most effective way to produce and disseminate knowledge with today’s technology? How can we best structure what we know and learn so that students, researchers, and scholars of the future can best understand the work of today’s researchers and scholars? I think that questions like these are more important and more difficult to address than whether we can put the contents of libraries on the Web.

Robert Sedgewick
December 10, 2007
Future of libraries?

1990 Every student spent significant time in the library
2010 Every student spends significant time online

Few faculty members in the sciences use the library at all for research

YET, the library’s budget continues to grow!

2020?
- A few book museums (for Grafton)
- Digital library infrastructure (for everyone else)

Scientific papers?

---

Alan Kay: “The best way to predict the future is to invent it.”
Scientific papers

When is the last time you visited a library to find a paper?
Did you print the papers to read the last time you refereed a conference?

Question: If it will not be read on paper, why write it as if it will?
Prediction: Someone will soon invent the future (should be easy)

"I could read it on my iPad ... if I had an iPad"
D. E. Knuth
Textbooks

A road to ruin

- prices continue to escalate
- students now rent, not own books
- planned obsolescence? walled garden?

Is there room for a good textbook?
Will free web resources prevail?
Sedgewick-Wayne publishing model

Two components

- traditional textbook (priced to own)
- forward-looking booksite (free)

Textbook

- traditional look-and-feel
- builds on 500 years of experience
- for use while learning

Booksight

- supports search
- has code, test data, animations
- links to references
- a living document
- for use while programming, exploring
Part I: Programming (2009)

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

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

3 Data Abstraction
   Data Types
   Creating DataTypes
   Designing Data Types
   Case Study: N-body

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

Part II: Computer science (in preparation)

5 A Computing Machine
   Data representations
   TOY machine
   Instruction Set
   Machine Language Coding
   Simulator

6 Building a Computer
   Boolean Logic and Gates
   Combinational Circuits
   Sequential Circuits
   TOY machine architecture

7 Theory of Computation
   Formal Languages
   Turing Machines
   Universality
   Computability
   Intractability

8 Systems
   Library Programming
   Compilers and Interpreters
   Operating Systems
   Networks
   Applications Systems

9 Scientific Computation
   Precision and Accuracy
   Differential Equations
   Linear Algebra
   Optimization
   Data Analysis
   Simulation
Textbook. Our textbook Introduction to Programming in Java [Amazon - Addison-Wesley] is an Interdisciplinary approach to the traditional CSI curriculum. We teach all of the classic elements of programming, using an "objects-in-the-middle" approach that emphasizes data abstraction. A key feature of the book is the manner in which we motivate each programming concept by examining its impact on specific applications, taken from fields ranging from materials science to genomics to astrophysics to Internet commerce. The book is organized around four stages of learning to program:

- Chapter 1: Elements of Programming introduces variables; assignment statements; built-in types of data; conditionals and loops; arrays; and input/output, including graphics and sound.
- Chapter 2: Functions introduces modular programming. We stress the fundamental idea of dividing a program into components that can be independently debugged, maintained, and reused.
- Chapter 3: Object-Oriented Programming introduces data abstraction. We emphasize the concept of a data type and its implementation using Java's class mechanism.
- Chapter 4: Algorithms and Data Structures introduces classical algorithms for sorting and searching, and fundamental data structures, including stacks, queues, and symbol tables.

Booksite. Reading a book and surfing the web are two different activities: This booksite is intended for your use while online (for example, while programming and while browsing the web); the textbook is for your use when initially learning new material and when reinforcing your understanding of that material (for example, when reviewing for an exam). The booksite consists of the following elements:

- Excerpts. A condensed version of the text narrative for reference while online.
- Exercises. Hundreds of exercises and some solutions.
- Java code. Hundreds of easily downloadable Java programs and real-world data sets.

To get started. Here are instructions for installing a Java programming environment [ Mac OS X · Windows · Linux ]. We also provide I/O libraries for reading and writing text and binary data, drawing graphics, and producing sound.

To adopt. Here are some of the distinctive features of our textbook and a marketing flyer. To preview our material, you can download the Preface and Chapter 1. If you wish to consider adoption, please fill out this form to request a copy of the textbook or ask for more information.

Last modified on February 65, 2012.

Copyright © 2002-2012 Robert Sedgewick and Kevin Wayne. All rights reserved.
Algorithms for the masses
Central thesis for *Algorithms* (1975)

All science/engineering students need an *algorithms* course

*Algorithms* embraces a significant body of knowledge that is

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

Barriers

- no room in curriculum
- need to implement all the algorithms (!)
- need to analyze all the algorithms (!)
- need to pick the most important ones
Current status of “Algorithms” (2012)

[Joint work with Kevin Wayne since 2007]

Any science/engineering student can appreciate

- data abstraction and modular programming
- 50+ classic and important algorithms and data structures
- historical context, applications
- relationships to OR, theory of algorithms

Algorithms (4th edition) and booksite (2011)

back to basics (one book)
Booksite

algs4.cs.princeton.edu

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

- Chapter 1: Fundamentals introduces a scientific and engineering basis for comparing algorithms and making predictions. It also includes our programming model.
- Chapter 2: Sorting considers several classic sorting algorithms, including insertion sort, mergesort, and quicksort. It also includes a binary heap implementation of a priority queue.
- Chapter 3: Searching describes several classic symbol table implementations, including binary search trees, red-black trees, and hash tables.
- Chapter 4: Graphs surveys the most important graph processing problems, including depth-first search, breadth-first search, minimum spanning trees, and shortest paths.
- Chapter 5: Strings investigates specialized algorithms for string processing, including radix sorting, substring search, tries, regular expressions, and data compression.
- Chapter 6: Context highlights connections to systems programming, scientific computing, commercial applications, operations research, and intractability.

Applications to science, engineering, and industry are a key feature of the text. We motivate each algorithm that we address by examining its impact on specific applications.

Booksite. Reading a book and surfing the web are two different activities: This booksite is intended for your use while online (for example, while programming and while browsing the web); the textbook is for your use when initially learning new material and when reinforcing your understanding of that material (for example, when reviewing for an exam). The booksite consists of the following elements:

- Excerpts. A condensed version of the text narrative, for reference while online.
- Java code. The algorithms and clients in this textbook.
- Exercise solutions. Solutions to selected exercises.

To get started. Here are instructions for setting up our recommended Java programming environment [ Mac OS X - Windows - Linux ].

To adopt. Here is a marketing flyer. Here is the preface. If you are considering adoption, you can ask the authors for more information or request an examination copy.

Last modified on September 18, 2011.

Copyright © 2002-2012 Robert Sedgewick and Kevin Wayne. All rights reserved.

- Text digests
- Ready-to-use code
- Supplementary exercises/answers
- Links to references and sources
- Modularized lecture slides
- Programming assignments
- Demos for lecture and precept
- Simulators for self-study
- Scientific applications
Top 100 algorithms

Algs4 code (and much more) all available online

Modular programming style
- one-click download
- test data
- variants
- robust library versions
- typical clients
Messages for algorithms students

Modern programming models are for you

Algorithms are important and useful in scientific, engineering, and commercial applications of all sorts

Performance matters

Classic algorithms for sorting, searching, graphs and strings have enabled the development of the computational infrastructure that surrounds us

A great many more important and useful algorithms remain to be discovered

Intrinsic limitations exist
Familiar and easy-to-motivate applications

Percolation

Ideal example/assignment
- teaches a basic CS concept
- solves an important problem
- intellectually engaging
- modular program
- is open-ended

union-find
Familiar and easy-to-motivate applications

Prim’s MST algorithm

Ideal example/assignment
• teaches a basic CS concept
• solves an important problem
• intellectually engaging
• modular program
• is open-ended

graph search
Familiar and easy-to-motivate applications

Bose-Einstein colliding particle simulation

Ideal example/assignment
- teaches a basic CS concept
- solves an important problem
- intellectually engaging
- modular program
- is open-ended

priority queue
Enrollments in algorithms course are also skyrocketing.

“Algorithms” enrollments now are three times what they were at the height of the bubble. At the time, enrollments seemed like a lot.

25+% of all Princeton students

Key factor in increase: All students in “CS for everyone” can take “Algorithms”
**Summary**

The scientific method is an essential ingredient in programming. Embracing, supporting, and leveraging science in intro CS and algorithms courses can serve large numbers of students.

Proof of concept: First-year courses at Princeton
- 50+% of Princeton students in a single intro course
- 25+% of Princeton students in a single algorithms course

Next goals:
- 50+% of all college students in an intro CS course
- 25+% of all college students in an algorithms course

**ALGORITHMS FOR THE MASSES**
Algorithms for the masses

Robert Sedgewick
Princeton University