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

**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

**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

**better experiment:**
- generate random numbers
- use $\chi^2$ 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

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

*textbook algorithm that flunks $\chi^2$ 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 \( \sqrt{\pi V/2} \)

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 \textit{st}-path in a graph

is a fundamental operation that demands understanding

\textbf{Ground rules for this talk}

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

\textbf{Applications}

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

\textbf{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?

<table>
<thead>
<tr>
<th></th>
<th>worst case upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortest</td>
<td>$VE/2$</td>
</tr>
<tr>
<td>max capacity</td>
<td>$2E \log C$</td>
</tr>
</tbody>
</table>

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?

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

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?

<table>
<thead>
<tr>
<th></th>
<th>worst case 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>
<td>37</td>
</tr>
<tr>
<td></td>
<td>( VC )</td>
<td>17,700</td>
<td></td>
</tr>
<tr>
<td>max capacity</td>
<td>( 2E \lg C )</td>
<td>26,575</td>
<td>7</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>Shortest</th>
<th>Worst case bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VE^2/2$</td>
<td>$VEC$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max capacity</th>
<th>Upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2E^2 \log C$</td>
<td></td>
</tr>
</tbody>
</table>

**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, $\log \log N$ 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

Ground rules

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

Ex: easy to find short paths quickly with \( A^* \) in geometric graphs (stay tuned)
Applications of grid graphs

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

<table>
<thead>
<tr>
<th>$M$</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>49</td>
<td>84</td>
</tr>
<tr>
<td>15</td>
<td>225</td>
<td>420</td>
</tr>
<tr>
<td>31</td>
<td>961</td>
<td>1860</td>
</tr>
<tr>
<td>63</td>
<td>3969</td>
<td>7812</td>
</tr>
<tr>
<td>127</td>
<td>16129</td>
<td>32004</td>
</tr>
<tr>
<td>255</td>
<td>65025</td>
<td>129540</td>
</tr>
<tr>
<td>511</td>
<td>261121</td>
<td>521220</td>
</tr>
</tbody>
</table>

Cost measure: number of graph edges examined
Finding an \textit{st}-path in a grid graph

Similar problems are covered extensively in the literature
\begin{itemize}
  \item Percolation
  \item Random walk
  \item Nonselfintersecting paths in grids
  \item Graph covering
\end{itemize}
\ldots

Elementary algorithms are found in textbooks
\begin{itemize}
  \item Depth-first search (DFS)
  \item Breadth-first search (BFS)
  \item Union-find
\end{itemize}
\ldots

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

Literature is no help, so
\begin{itemize}
  \item Implement elementary algorithms
  \item Use scientific method to study performance
\end{itemize}
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)

**GRAPH API**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAPH(Edge[] a)</td>
<td>construct a GRAPH from an array of edges</td>
</tr>
<tr>
<td>void findPath(int s, int t)</td>
<td>conduct a search from s to t</td>
</tr>
<tr>
<td>int st(int v)</td>
<td>return predecessor of v on path found</td>
</tr>
</tbody>
</table>

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

**Client code for grid graphs**

```java
public class GRAPH
{
    GRAPH(Edge[] a)
    void findPath(int s, int t)
    int st(int v)
}
```

```
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);
```
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 $Q$

if $x = t$

done

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**

```java
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]);
    }
```

**DFS implementation (code to save path omitted)**

```java
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);
    }
```

**graph representation**

- **vertex-indexed array of linked lists**
- **two nodes per edge**
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

\[ \sim E/2 \]

south, north, east, west

\[ \sim E^{1/2} \]

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**

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

**randomized iterator**

```java
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];
    }
```

represent graph with arrays, not lists

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**

```java
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;
}
```

**DFS implementation (code to save path omitted)**

```java
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);
}
```

**graph representation**

vertex-indexed array of variable-length arrays
BFS: standard implementation

Use a queue to hold fringe vertices

```
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; }
        }
    }
}
```

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?
Experimental results

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

<table>
<thead>
<tr>
<th>M</th>
<th>V</th>
<th>E</th>
<th>BFS</th>
<th>DFS</th>
<th>UF</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>49</td>
<td>168</td>
<td>0.75</td>
<td>0.32</td>
<td>1.05</td>
</tr>
<tr>
<td>15</td>
<td>225</td>
<td>840</td>
<td>0.75</td>
<td>0.45</td>
<td>1.02</td>
</tr>
<tr>
<td>31</td>
<td>961</td>
<td>3720</td>
<td>0.75</td>
<td>0.36</td>
<td>1.14</td>
</tr>
<tr>
<td>63</td>
<td>3969</td>
<td>15624</td>
<td>0.75</td>
<td>0.32</td>
<td>1.05</td>
</tr>
<tr>
<td>127</td>
<td>16129</td>
<td>64008</td>
<td>0.75</td>
<td>0.40</td>
<td>0.99</td>
</tr>
<tr>
<td>255</td>
<td>65025</td>
<td>259080</td>
<td>0.75</td>
<td>0.42</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Analytic proof?
Faster algorithms available?
A standard search paradigm
gives a faster algorithm or finding an \textit{st}-path in a graph

Use \textbf{two} depth-first searches
\begin{itemize}
\item one from the source
\item one from the destination
\item interleave the two
\end{itemize}

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

Examines 13\% of the edges
\textbf{3-8 times} faster than standard implementations

Faster approach?
Other models?

\textbf{Not log log N, but not bad!}
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

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

How does 2-way DFS do in this model?

Experiment:

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

Surprising result: Finds short paths in $\sim E^{1/2}$ 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: $2E^{1/2}$ steps $\sim 2V^{1/2}$ random vertices
   - step 2: like birthday paradox

Path length?
Multiple searchers revisited?

Next steps: refine model, more experiments, detailed proofs.
Detailed example: paths in graphs

End of “lecture-within-a-lecture”

<table>
<thead>
<tr>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Randomization makes cost depend on graph, not representation.</td>
</tr>
<tr>
<td>• DFS is faster than BFS or UF for finding paths in grid graphs.</td>
</tr>
<tr>
<td>• Two DFSs are faster than 1 DFS — or N of them — in grid graphs.</td>
</tr>
<tr>
<td>• We can find short paths quickly in small-world graphs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What are the BFS, UF, and DFS constants in grid graphs?</td>
</tr>
<tr>
<td>• Is there a sublinear algorithm for grid graphs?</td>
</tr>
<tr>
<td>• Which methods adapt to directed graphs?</td>
</tr>
<tr>
<td>• Can we precisely analyze and quantify costs for small-world graphs?</td>
</tr>
<tr>
<td>• What is the cost distribution for DFS for any interesting graph model?</td>
</tr>
<tr>
<td>• How effective are these methods for other graph families?</td>
</tr>
<tr>
<td>• Do these methods lead to faster maxflow algorithms?</td>
</tr>
<tr>
<td>• How effective are these methods in practice?</td>
</tr>
</tbody>
</table>

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

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

\[ B(z) = B_0 z^0 + B_1 z^1 + B_2 z^2 + B_3 z^3 + \ldots \]

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

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

that we can solve algebraically

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

and expand to get coefficients

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

that we can approximate

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

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**

\[
<G> = \varepsilon + <G> + <G> \times <G> + <G> \times <G> \times <G> + \ldots
\]

**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 \( G(z) = \frac{1}{1 - G(z)} \),
so \( G(z)^2 - G(z) + z = 0 \)

and treat as a complex function to approximate growth:

\[
G_N \sim \frac{4^N}{2N \Gamma(\frac{1}{2}) \sqrt{N}} = \frac{4^N}{2N \sqrt{\pi N}}
\]

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:*

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

**Cauchy coefficient formula**

**Hankel contour**

**many details omitted!**
Analytic combinatorics: universal laws

of sweeping generality derive from the same technology

Ex. Context free constructions

< G_0 > = \text{OP}_0(< G_0 >, < G_1 >, \ldots, < G_t >)
< G_1 > = \text{OP}_1(< G_0 >, < G_1 >, \ldots, < G_t >)
\ldots
< G_t > = \text{OP}_t(< G_0 >, < G_1 >, \ldots, < G_t >)

\text{Combinatorial constructions}

\text{directly map to a system of GFs}

\text{that we can manipulate algebraically to get a single complex function}

\text{that is amenable to singularity analysis}

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

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:

- \( \lg(T(2N)/T(N)) \rightarrow c \) \( \text{the doubling test that is the basis for predicting performance!} \)
- \( T(N)/b^N \ N^c \rightarrow a \)

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

any program you might want to write

- data abstraction
  - functions and modules
    - graphics, sound, and image I/O
      - arrays
        - conditionals and loops
          - Math
          - text I/O
            - assignment statements
              - primitive types

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

<table>
<thead>
<tr>
<th><strong>functions</strong></th>
<th><strong>sin() cos(), log()</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>libraries</strong></td>
<td>I/O, data analysis</td>
</tr>
<tr>
<td><strong>1D arrays</strong></td>
<td>sound</td>
</tr>
<tr>
<td><strong>2D arrays</strong></td>
<td>images</td>
</tr>
<tr>
<td><strong>strings</strong></td>
<td>genomes</td>
</tr>
<tr>
<td><strong>object-oriented I/O</strong></td>
<td>streaming from the web</td>
</tr>
<tr>
<td><strong>OOP</strong></td>
<td>Brownian motion</td>
</tr>
<tr>
<td><strong>data structures</strong></td>
<td>small-world phenomenon</td>
</tr>
</tbody>
</table>
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 (continued)

Standard enrollment pattern

1995

- Humanities
- Social Sciences
- Science/Math
- Engineering

-ab advising system
-computing requirement

intro CS
Progress report (continued)

Standard enrollment pattern (up and down)

- Humanities
- Social Sciences
- Science/Math
- Engineering

2001

AB advising system
computing requirement

intro CS

2001
Standard enrollment pattern (up and down), but now is skyrocketing

40% of all Princeton students
Elective for 40% of enrollees
40% female
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

www.cs.princeton.edu/introcs
**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
Introduction to Computer Science
R. Sedgewick and K. Wayne

**Prologue**

**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

**A Computing Machine**
- Data representations
- TOY machine
- Instruction Set
- Machine-Language Programming
- Simulator

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

**Theory of Computation**
- Formal Languages and Machines
- Turing Machines
- Universality
- Computability
- Intractability

**Systems**
- Library Programming
- Compilers, Interpreters, and Emulators
- Operating Systems
- Networks
- Applications Systems

**Scientific Computation**
- Precision and Accuracy
- Differential Equations
- Linear Algebra
- Optimization
- Data Analysis
- Simulation
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

```
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 simulation

efficient algorithms are necessary

a poster child for priority queue abstraction
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 student's 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!