Performance

CS 121: Data Structures

Putting Performance in Perspective

- When developing a program, always prioritize **readability, correctness, and maintainability**
- Not all programs have strict performance constraints
- But if performance is important:
  - Focus on big-picture issues throughout development (e.g., selection of appropriate libraries, data structures, algorithms, etc.)
  - Only spend time on small, tedious optimizations if benchmarking has identified a particular area of code as performance-limiting

Big-Picture Performance Issue: Algorithms

- Recursion vs dynamic programming makes a big difference when computing the Fibonacci sequence!
- Think about:
  - Is your program performing the same calculation repeatedly?
  - Is your program storing data it doesn’t need?

Big-Picture Performance Issue: Algorithms

FibonacciR.java

```
public class FibonacciR {
  public static long F(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return F(n-1) + F(n-2);
  }
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    StdOut.println(F(n));
  }
}
```

```
> time java-introcs FibonacciR 50
real    0m54.950s
```

~40 billion calls to F(), ~20 billion sums performed

FibonacciD.java

```
public class FibonacciD {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    long[] F = new long[n + 1];
    F[0] = 0;
    F[1] = 1;
    for (int i = 2; i <= n; i++)
      F[i] = F[i - 1] + F[i - 2];
    StdOut.println(F[n]);
  }
}
```

```
> time java-introcs FibonacciD 50
real    0m0.115s
```

51 sums performed

Big-Picture Performance Issue: Algorithms

FibonacciD.java

```
public class FibonacciD {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    long[] F = new long[n + 1];
    F[0] = 0;
    F[1] = 1;
    for (int i = 2; i <= n; i++)
      F[i] = F[i - 1] + F[i - 2];
    StdOut.println(F[n]);
  }
}
```

```
> time java-introcs FibonacciD 50
real     0m0.115s
```

Fibonacci.java

```
public class Fibonacci {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    long nMinus2 = 0; long nMinus1 = 1;
    long nMinus0 = 1;
    for (int i = 2; i <= n; i++) {
      nMinus0 = nMinus1 + nMinus2;
      nMinus2 = nMinus1; nMinus1 = nMinus0;
    }
    StdOut.println(nMinus0);
  }
}
```

```
> time java-introcs Fibonacci 50
real     0m0.100s
```

Note: To keep this program short, it doesn’t work for n=0

Big-Picture Performance Issue: Streaming Data

- Arbitrary-sized data should be processed incrementally, when possible
- If you try to store all data in memory at once, your program can easily run out of memory!

Big-Picture Performance Issue: Streaming Data

```
# Generate a 1 million line input file (~10MB)
> cat /dev/random | \
  LC_ALL=C tr -dc 'a-zA-Z0-9' | \
  fold -w 10 | \
  head -n 1000000 > 1m.txt

# Processing the file line-by-line works great!
> java -Xmx32M Grep hello 1m.txt

# We run out of memory if we use readAllLines()
> java -Xmx32M GrepAll hello 1m.txt
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.regex.Matcher.usePattern(Matcher.java:381)
        at java.base/java.util.Scanner.findPatternInBuffer(Scanner.java:1080)
        at java.base/java.util.Scanner.findWithinHorizon(Scanner.java:1791)
        at java.base/java.util.Scanner.hasNextLine(Scanner.java:1610)
        at In.hasNextLine(In.java:248)
        at In.readAllLines(In.java:528)
        at GrepAll.main(GrepAll.java:11)
```

Note: We limited Java to 32MB of memory with the -Xmx32M argument.

Big-Picture Performance Issue: Memory Impact of Language

- Python, Java, JavaScript, etc. use automatic **“garbage collection”** to manage memory
- In contrast, C, C++, Objective-C, etc. are primarily used with **manual memory management**
  - Rust, Swift, etc. support something in between, that gives benefits of both (e.g., automatic reference counting)
- Garbage collection requires more memory, and has performance overhead. However, manual memory management requires more effort from developers.

Big-Picture Performance Issue: Memory Impact of Language

"These results quantify the time-space tradeoff of garbage collection: with five times as much memory [garbage collection] matches the performance of reachability-based explicit memory management. With only three times as much memory, the collector runs on average 17% slower than explicit memory management. However, with only twice as much memory, garbage collection degrades performance by nearly 70%. **When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management.**"

Hertz, Matthew, and Emery D. Berger. "Quantifying the performance of garbage collection vs. explicit memory management." Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 2005.

Big-Picture Performance Issue: Garbage Collection on Smartphones

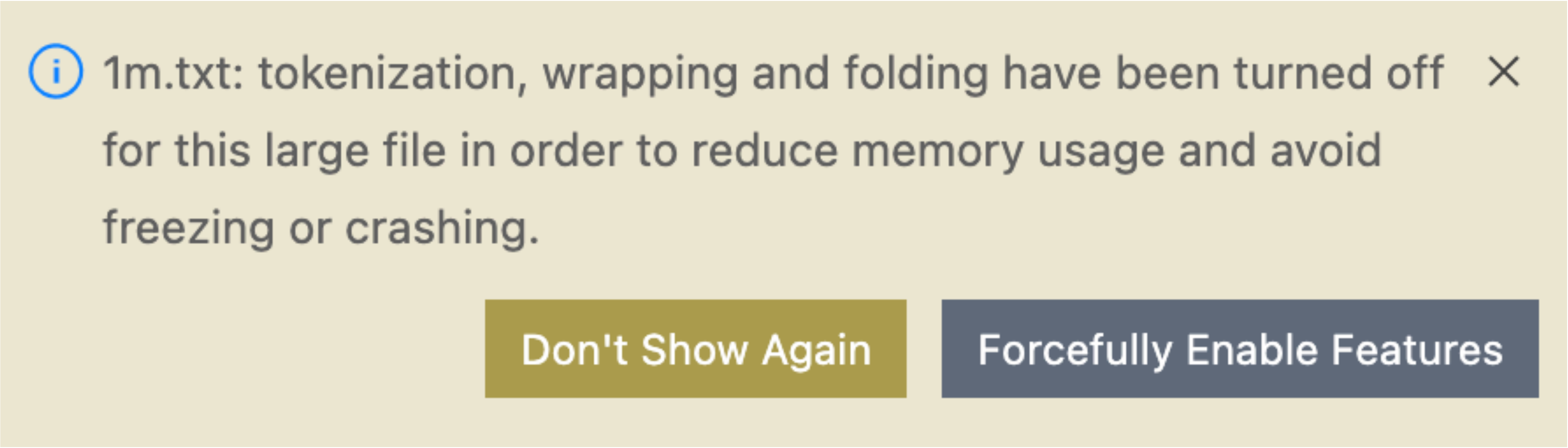
- iOS apps are coded in Objective-C and Swift, and **don't use garbage collection**
- Android apps are written in Java and Kotlin, and **use garbage collection**
- This means that Android phones need more RAM to give comparable performance

iPhone 14
6GB of RAM

Samsung Galaxy S22 Ultra 5G
8GB or 12GB of RAM

Big-Picture Performance Issue: Garbage Collection on the Desktop

- Visual Studio Code is written using the "Electron" framework, which includes a customized Chrome web browser that runs code written in JavaScript (JavaScript uses garbage collection)
    - Writing "for the web" makes it easier to maintain VS Code on different platforms
    - However, Electron-based apps use much more memory than "native" apps (e.g., Sublime Text, Notepad++, BBEdit, etc.). This is visible when trying to open a large file in VS Code:



Memory usage with one small file open:

- Sublime Text: 67MB
- BBEdit: 121MB
- VS Code: 842MB
- IntelliJ: 1.17GB

Optimizing Matrix Multiplication

- In scientific computing, matrix multiplication is common
- How to perform matrix multiplication most efficiently?
- Subtle differences in implementation can have a large impact on performance

Basic Matrix Multiplication

MatrixMult.java

```
StopwatchCPU timer = new StopwatchCPU();
double[][] c = new double[N][N];
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
StdOut.printf("Multiplied in %.2f seconds\n",
              timer.elapsedTime());
```

```
> java-introcs MatrixMult 1000
Multiplied in 6.24 seconds
```

Performance-Tuned Matrix Multiplication

MatrixMult.java

```
StopwatchCPU timer = new StopwatchCPU();
double[][] c = new double[N][N];
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
StdOut.printf("Multiplied in %.2f seconds\n",
              timer.elapsedTime());
```

```
> java-introcs MatrixMult 1000
Multiplied in 6.24 seconds
```

MatrixMultAlt.java

```
StopwatchCPU timer = new StopwatchCPU();
double[][] c = new double[N][N];
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
StdOut.printf("Multiplied in %.2f seconds\n",
              timer.elapsedTime());
```

```
> java-introcs MatrixMultAlt 1000
Multiplied in 1.36 seconds
```

Faster, but why? Will it always be faster?

Using a Linear Algebra Library

MatrixMultAlt.java

```
StopwatchCPU timer = new StopwatchCPU();
double[][] c = new double[N][N];
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
StdOut.printf("Multiplied in %.2f seconds\n",
              timer.elapsedTime());
```

```
> java-introcs MatrixMultAlt 1000
Multiplied in 1.36 seconds
```

MatrixMultJblas.java

```
// Initialize the matrices to random values
DoubleMatrix a = DoubleMatrix.randn(N, N);
DoubleMatrix b = DoubleMatrix.randn(N, N);

// Perform matrix multiplication
StopwatchCPU timer = new StopwatchCPU();
DoubleMatrix c = a.mmul(b);
StdOut.printf("Multiplied in %.2f seconds\n",
              timer.elapsedTime());
```

```
> java MatrixMultJblas 1000
Multiplied in 0.21 seconds
```

Less code, and should consistently be faster.

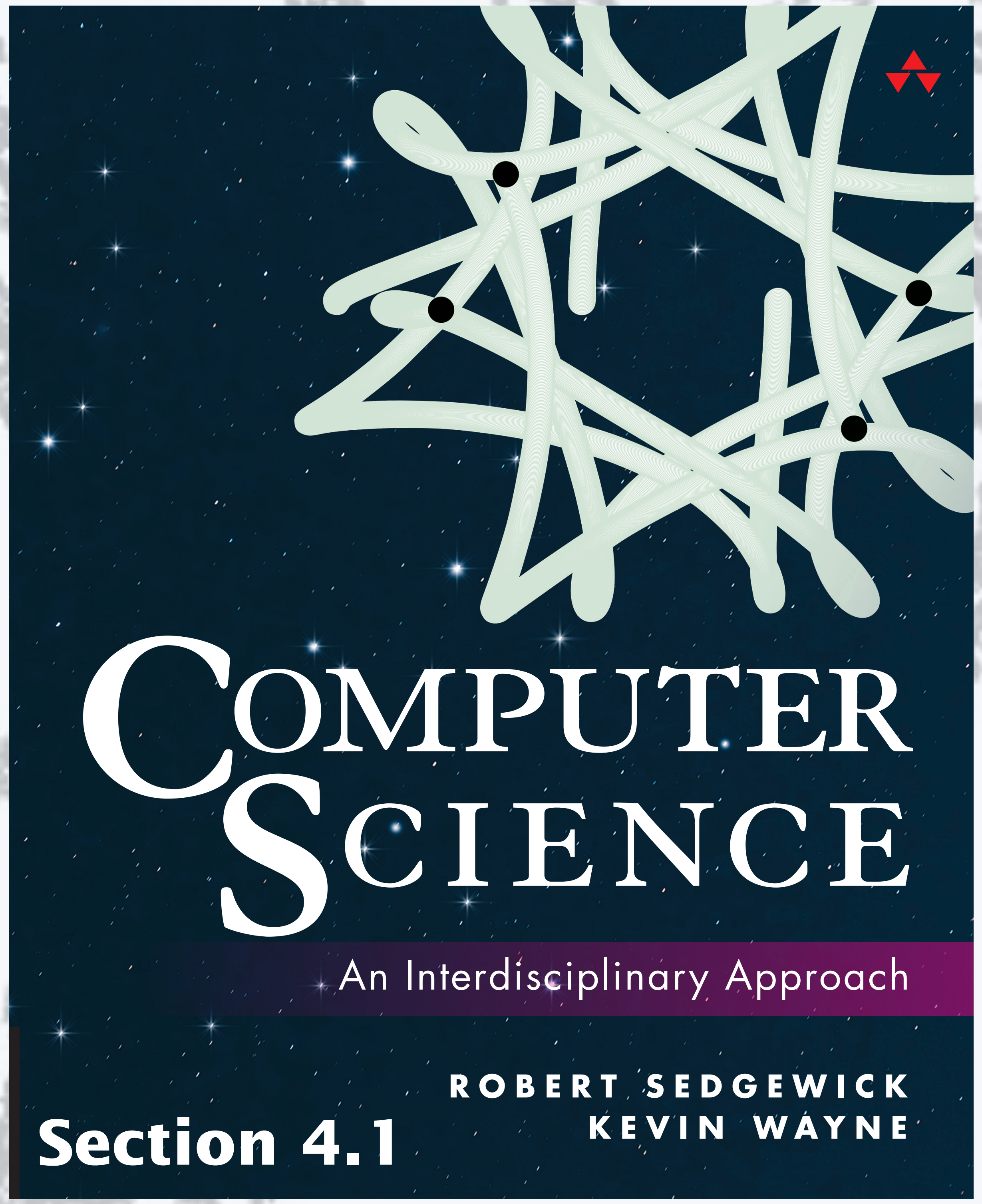
Takeaways

- 97% of the time, you should **think about performance at a high-level**
- High-level, conceptual thinking will help you select the appropriate:
  - Language
  - Libraries
  - Algorithms
  - Data structures
- Don’t spend time on tedious performance tuning (e.g., swapping the order of for-loops), unless you are creating a reusable, high-performance library!

COMPUTER SCIENCE

SEDGEWICK / WAYNE

PART I: PROGRAMMING IN JAVA



http://introcs.cs.princeton.edu

7. Performance

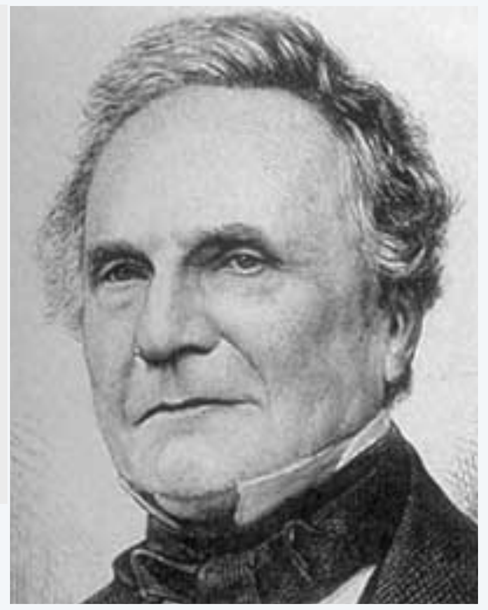
7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

The challenge (since the earliest days of computing machines)

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

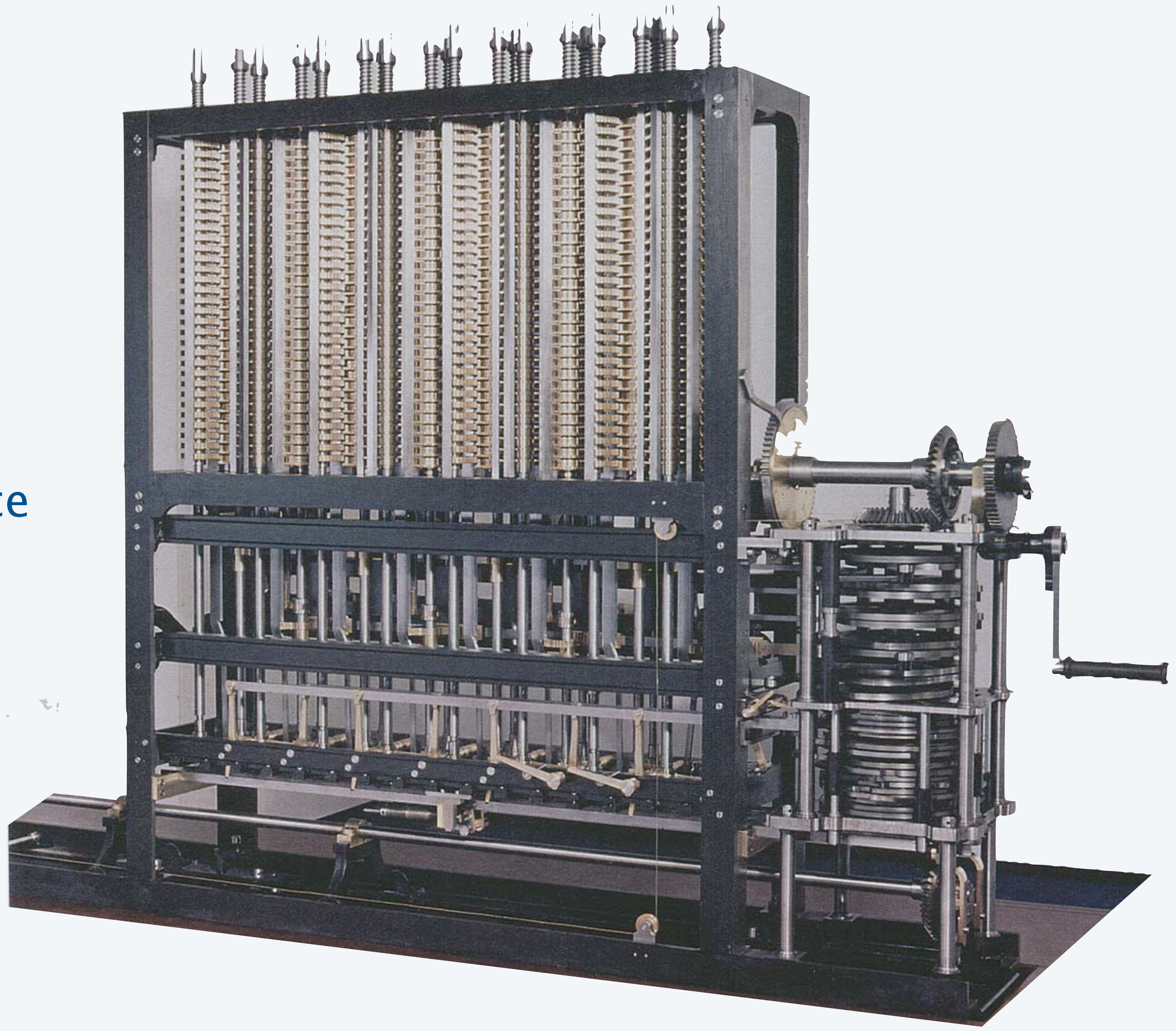
– Charles Babbage



Difference Engine #2

Designed by Charles Babbage, c. 1848

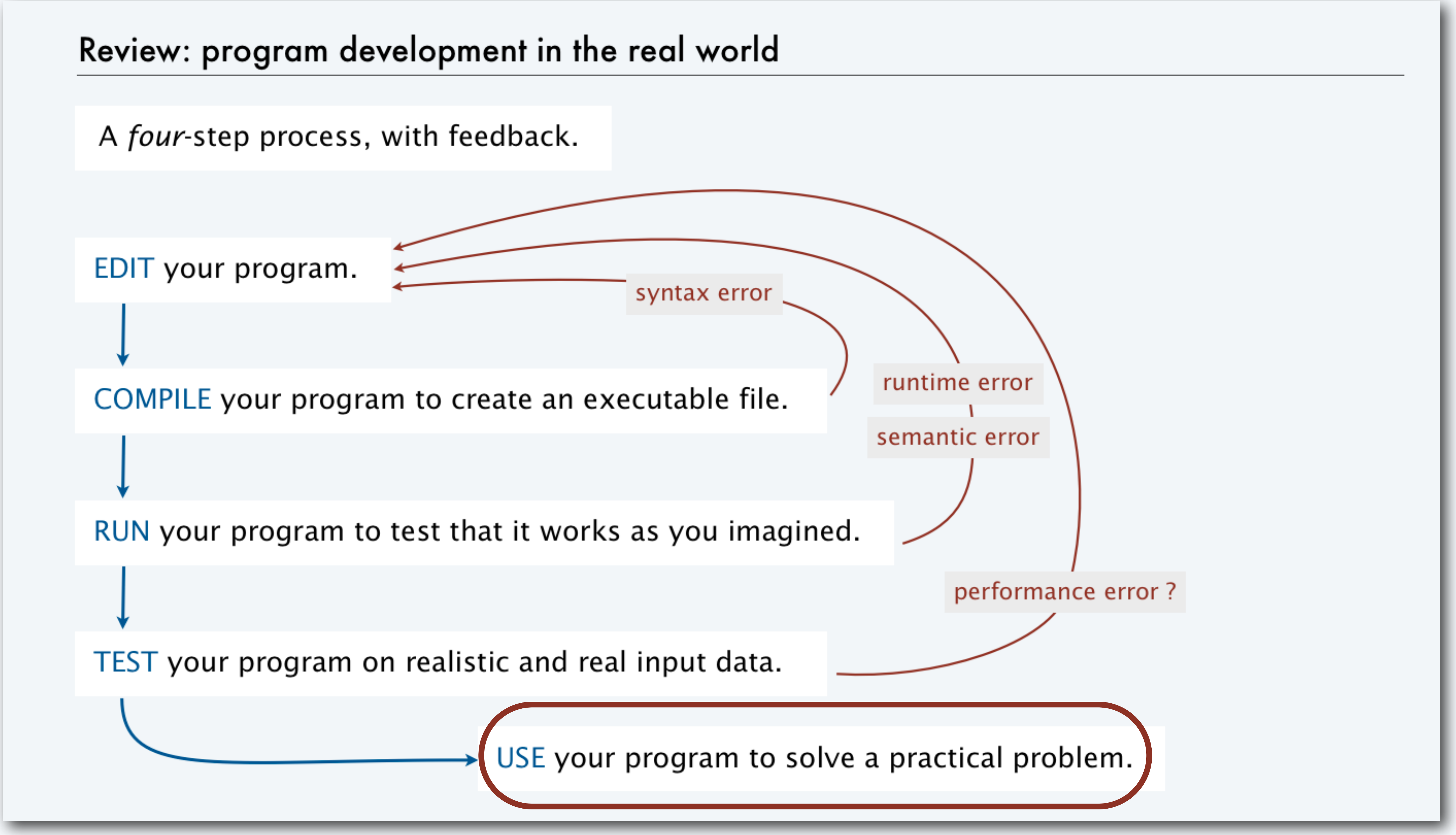
Built by London Science Museum, 1991



Q. How many times do you have to turn the crank?

The challenge (modern version)

Q. Will I be able to use my program to solve a large practical problem?



Q. If not, how might I understand its performance characteristics so as to improve it?

Key insight (Knuth 1970s). Use the *scientific method* to understand performance.

Three reasons to study program performance

1. To predict program behavior

- Will my program finish?
- *When* will my program finish?

2. To compare algorithms and implementations.

- Will this change make my program faster?
- How can I make my program faster?

3. To develop a basis for understanding the problem and for designing new algorithms

- Enables new technology.
- Enables new research.

```
public class Gambler
{
    public static void main(String[] args)
    {
      int stake  = Integer.parseInt(args[0]);
      int goal   = Integer.parseInt(args[1]);
      int trials = Integer.parseInt(args[2]);
      int wins   = 0;
      for (int t = 0; t < trials; t++)
      {
         int cash = stake;
         while (cash > 0 && cash < goal)
            if (Math.random() < 0.5) cash++;
            else                     cash--;
         if (cash == goal) wins++;
      }
      StdOut.print(wins + " wins of " + trials);
   }
}
```

An *algorithm* is a method for solving a problem that is suitable for implementation as a computer program.

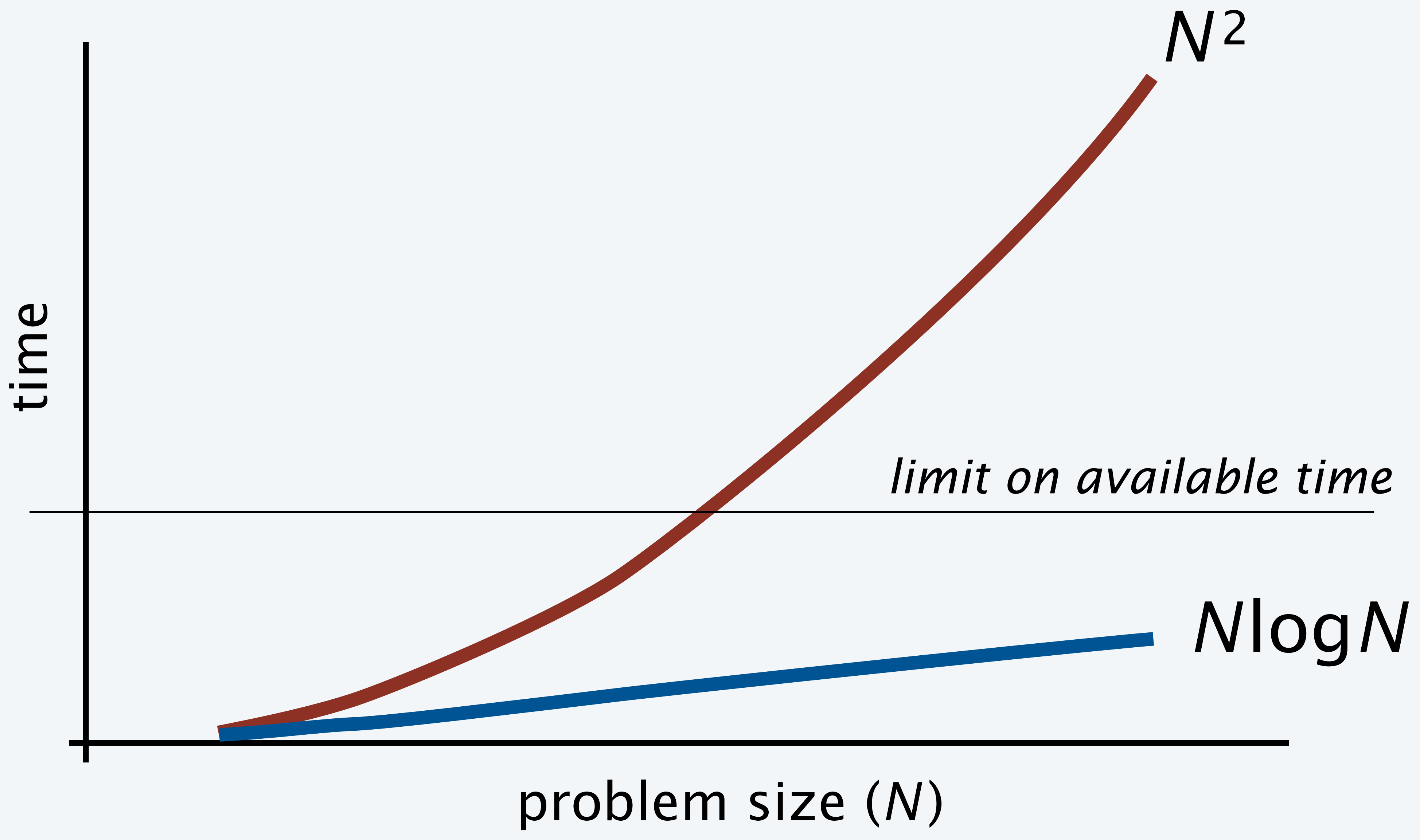
We study several algorithms later in this course.
Taking more CS courses? You'll learn dozens of algorithms.

An algorithm design success story

N-body simulation

- Goal: Simulate gravitational interactions among N bodies.
- Brute-force algorithm uses N^2 steps per time unit.
- Issue (1970s): Too slow to address scientific problems of interest.
- Success story: *Barnes-Hut* algorithm uses $N \log N$ steps and *enables new research*.

Andrew Appel
PU '81
senior thesis

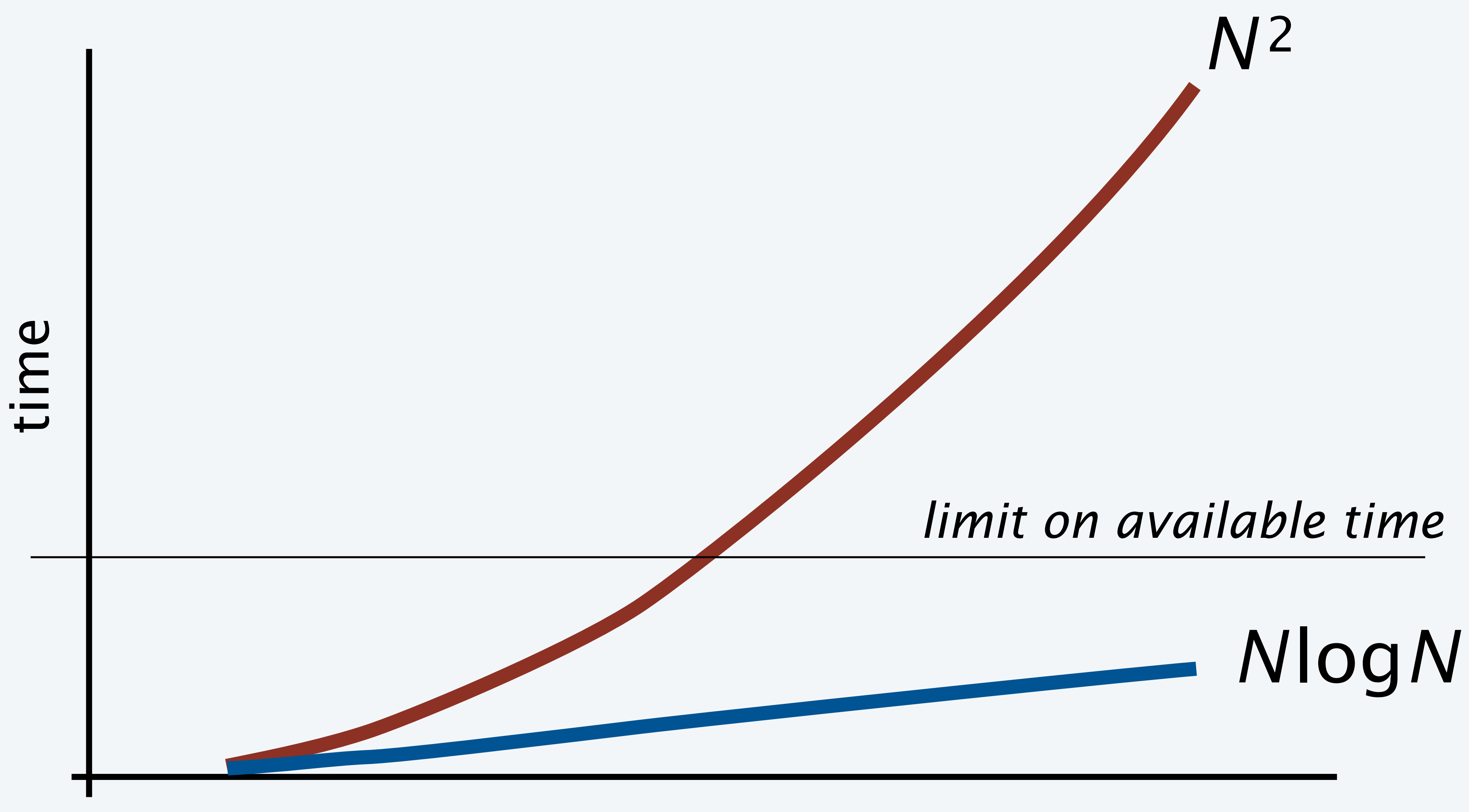


Another algorithm design success story

Discrete Fourier transform

- Goal: Break down waveform of *N* samples into periodic components.
- Applications: digital signal processing, spectroscopy, …
- Brute-force algorithm uses N^2 steps.
- Issue (1950s): Too slow to address commercial applications of interest.
- Success story: *FFT* algorithm uses $N \log N$ steps and *enables new technology*.

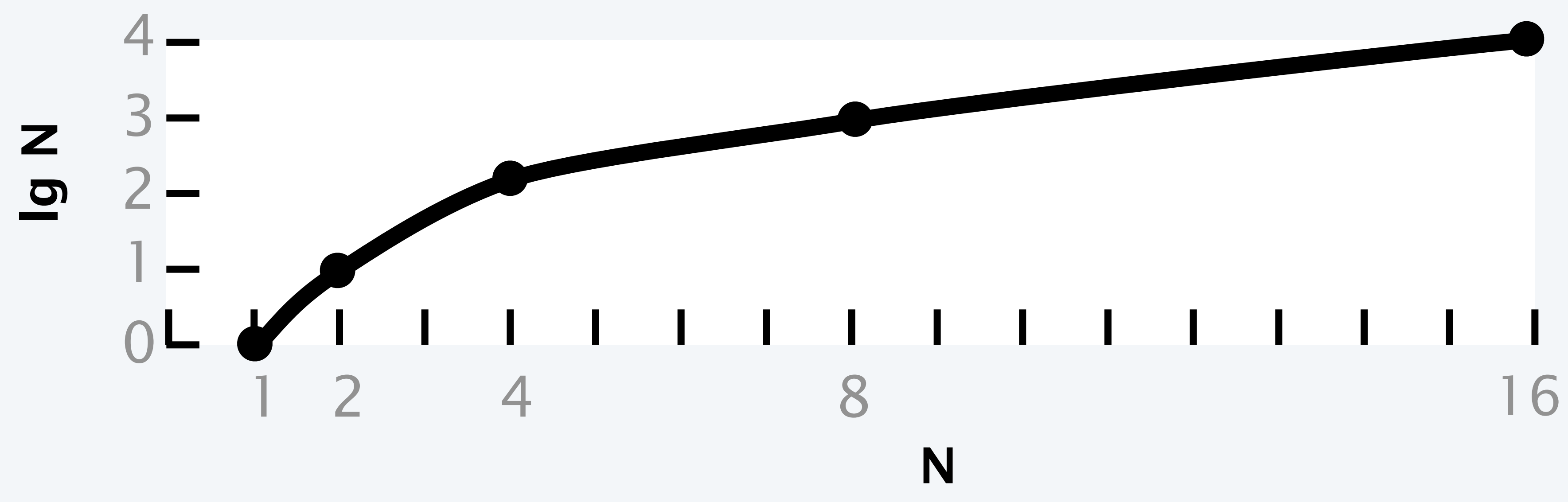
John Tukey
1915–2000



Quick aside: binary logarithms

Def. The *binary logarithm* of a number N (written $\lg N$) is the number x satisfying $2^x = N$.

or $\log_2 N$



Frequently encountered values

N	*approximate value*	$\lg N$	$\log_{10} N$
2^{10}	1 thousand	10	3.01
2^{20}	1 million	20	6.02
2^{30}	1 billion	30	9.03

Q. How many recursive calls for `convert(N)`?

```
public static String convert(int N)
{
   if (N == 1) return "1";
   return convert(N/2) + (N % 2);
}
```

A. Largest integer less than or equal to $\lg N$ (written $\lfloor \lg N \rfloor$).

Prove by induction.
Details in "sorting and searching" lecture.

Fact. The number of bits in the binary representation of N is $1 + \lfloor \lg N \rfloor$.

Fact. Binary logarithms arise in the study of algorithms based on recursively solving problems half the size (*divide-and-conquer algorithms*), like convert, FFT and Barnes-Hut.

An algorithmic challenge: 3-sum problem

Three-sum. Given *N* integers, enumerate the triples that sum to 0.

For simplicity, just count them.

```
public class ThreeSum
{
   public static int count(int[] a)
   {  /* See next slide. */ }

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

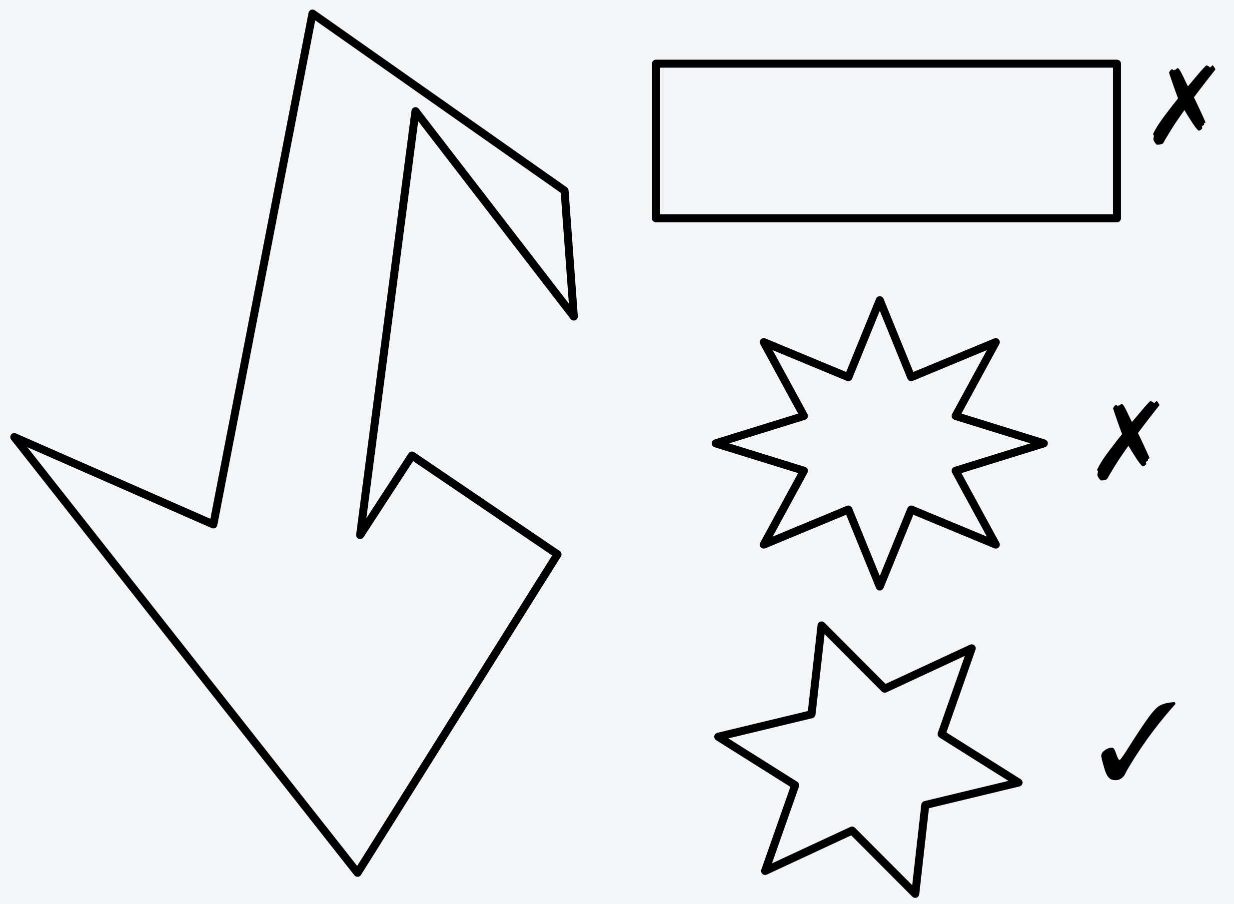
```
% more 6ints.txt
30 -30 -20 -10 40  0

% java ThreeSum <  6ints.txt
3
```

30	-30	0
30	-20	-10
-30	-10	40

Applications in computational geometry

- Find collinear points.
- Does one polygon fit inside another?
- Robot motion planning.
- [a surprisingly long list]



Q. Can we solve this problem for $N = 1$ million?

Three-sum implementation

"Brute force" algorithm

- Process all possible triples.
- Increment counter when sum is 0.

i	0	1	2	3	4	5
a[i]	30	-30	-20	-10	40	0

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 0;
   for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
         for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
               cnt++;
   return cnt;
}
```

Keep i < j < k to avoid processing each triple 6 times

$\binom{N}{3}$ triples with i < j < k

i	j	k	a[i]	a[j]	a[k]
0	1	2	30	-30	-20
0	1	3	30	-30	-10
0	1	4	30	-30	40
0	1	5	30	-30	0
0	2	3	30	-20	-10
0	2	4	30	-20	40
0	2	5	30	-20	0
0	3	4	30	-10	40
0	3	5	30	-10	0
0	4	5	30	40	0
1	2	3	-30	-20	-10
1	2	4	-30	-20	40
1	2	5	-30	-20	0
1	3	4	-30	-10	40
1	3	5	-30	-10	0
1	4	5	-30	40	0
2	3	4	-20	-10	40
2	3	5	-20	-10	0
2	4	5	-20	40	0
3	4	5	-10	40	0

Q. How much time will this program take for $N = 1$ million?

Image sources

http://commons.wikimedia.org/wiki/File:Babbages_Analytical_Engine,_1834-1871._(9660574685).jpg
http://commons.wikimedia.org/wiki/File:Charles_Babbage_1860.jpg
http://commons.wikimedia.org/wiki/File:John_Tukey.jpg
http://commons.wikimedia.org/wiki/File:Andrew_Apple_(FloC_2006).jpg
http://commons.wikimedia.org/wiki/File:Hubble's_Wide_View_of_'Mystic_Mountain'_in_Infrared.jpg

7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

A first step in analyzing running time

Find representative inputs

- Option 1: Collect actual input data.
- Option 2: Write a program to generate representative inputs.

Input generator for ThreeSum

```
public class Generator
{  // Generate N integers in [-M, M)
   public static void main(String[] args)
   {
      int M = Integer.parseInt(args[0]);
      int N = Integer.parseInt(args[1]);
      for (int i = 0; i < N; i++)
         StdOut.println(StdRandom.uniform(-M, M));
   }
}
```

```
% java Generator 1000000 10
28773
-807569
-425582
594752
600579
-483784
-861312
-690436
-732636
360294
```

not much chance of a 3-sum

```
% java Generator 10 10
-2
1
-4
1
-2
-10
-4
1
0
-7
```

good chance of a 3-sum

Empirical analysis

Run experiments

- Start with a moderate input size *N*.
- Measure and record running time.
- Double input size *N*.
- Repeat.
- Tabulate and plot results.

Measure running time

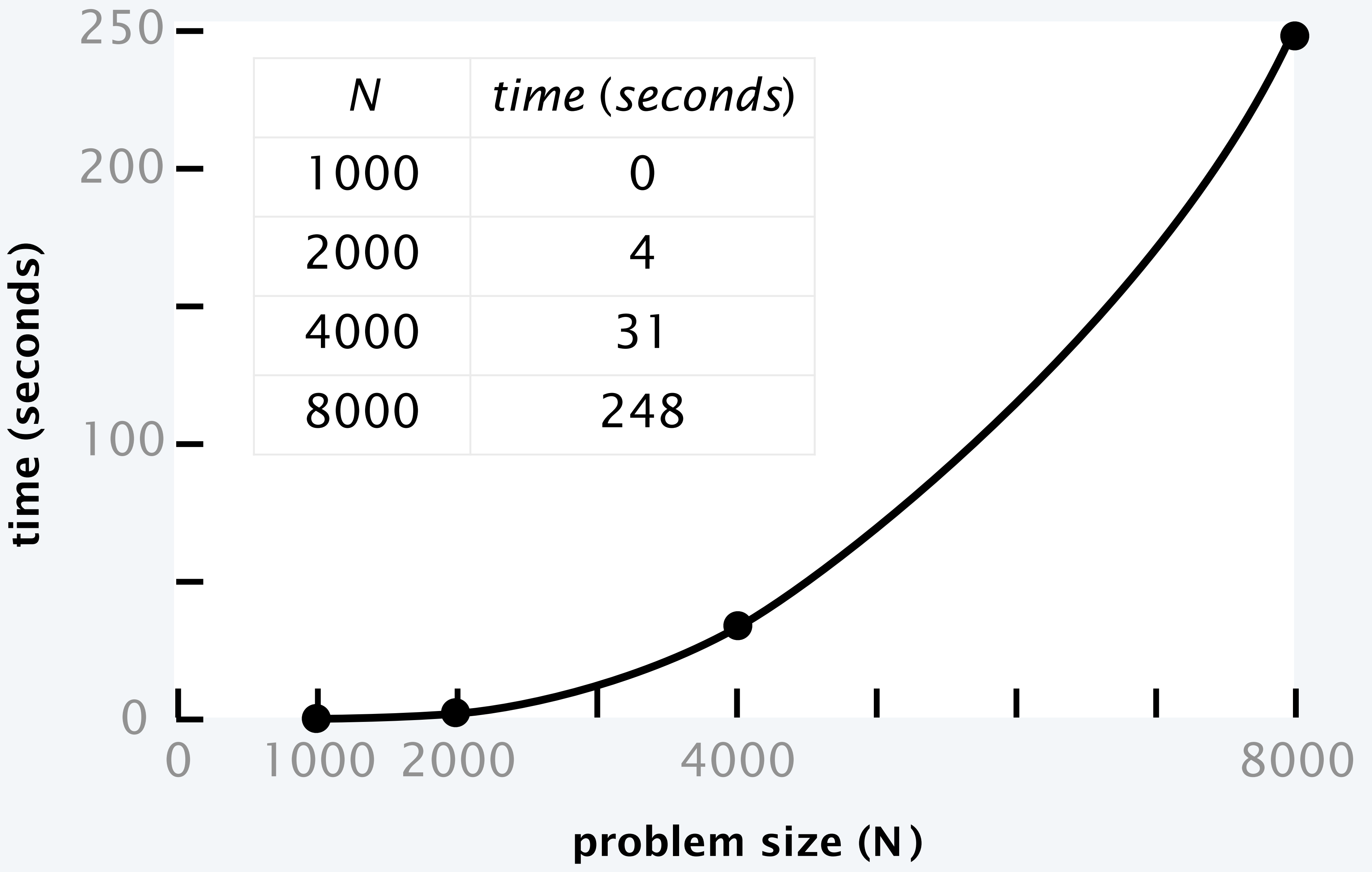
Replace println() in ThreeSum with this code.

```
double start = System.currentTimeMillis() / 1000.0;
int cnt = count(a);
double now   = System.currentTimeMillis() / 1000.0;
StdOut.printf("%d (%.0f seconds)\n", cnt, now - start);
```

Run experiments

```
% java Generator 1000000 1000 | java ThreeSum
59 (0 seconds)
% java Generator 1000000 2000 | java ThreeSum
522 (4 seconds)
% java Generator 1000000 4000 | java ThreeSum
3992 (31 seconds)
% java Generator 1000000 8000 | java ThreeSum
31903 (248 seconds)
```

Tabulate and plot results



N	*time (seconds)*
1000	0
2000	4
4000	31
8000	248

Aside: experimentation in CS

is *virtually free*, particularly by comparison with other sciences.

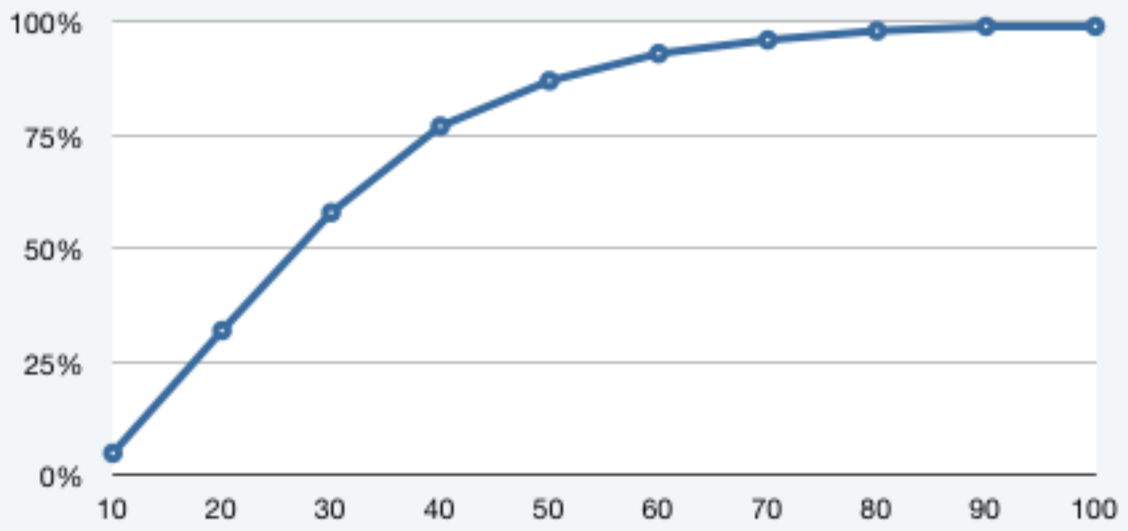
one million experiments →

```
% java SelfAvoidingWalker 10 100000
5% dead ends
% java SelfAvoidingWalker 20 100000
32% dead ends
% java SelfAvoidingWalker 30 100000
58% dead ends
% java SelfAvoidingWalker 40 100000
77% dead ends
% java SelfAvoidingWalker 50 100000
87% dead ends
% java SelfAvoidingWalker 60 100000
93% dead ends
% java SelfAvoidingWalker 70 100000
96% dead ends
% java SelfAvoidingWalker 80 100000
98% dead ends
% java SelfAvoidingWalker 90 100000
99% dead ends
% java SelfAvoidingWalker 100 100000
99% dead ends
```

Chemistry

one experiment

Physics



Computer Science

Bottom line. *No excuse* for not running experiments to understand costs.

Data analysis

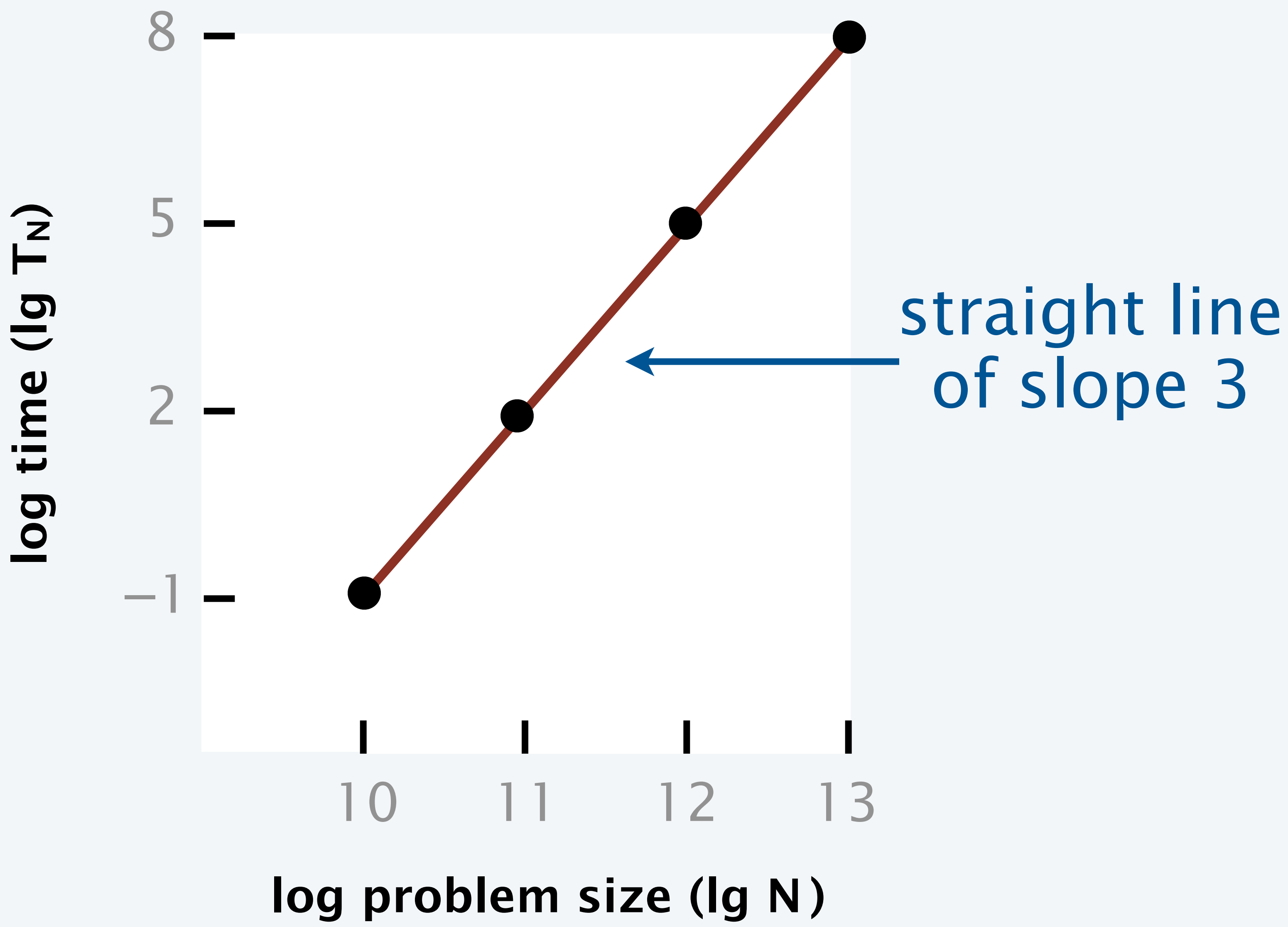
Curve fitting

- Plot on *log-log scale*.
- If points are on a straight line (often the case), a *power law* holds—a curve of the form aN^b fits.
- The exponent b is the slope of the line.
- Solve for a with the data.

N	T_N	$\lg N$	$\lg T_N$	$4.84 \times 10^{-10} \times N^3$
1000	0.5	10	−1	0.5
2000	4	11	2	4
4000	31	12	5	31
8000	248	13	8	248

✓

log–log plot



Do the math

x-intercept (use lg in anticipation of next step)

$\lg T_N = \lg a + 3 \lg N$	equation for straight line of slope 3
$T_N = aN^3$	raise 2 to a power of both sides
$248 = a \times 8000^3$	substitute values from experiment
$a = 4.84 \times 10^{-10}$	solve for a
$T_N = 4.84 \times 10^{-10} \times N^3$	substitute

a curve that fits the data ?

Prediction and verification

Hypothesis. Running time of ThreeSum is $4.84 \times 10^{-10} \times N^3$.

Prediction. Running time for $N = 16{,}000$ will be 1982 seconds.

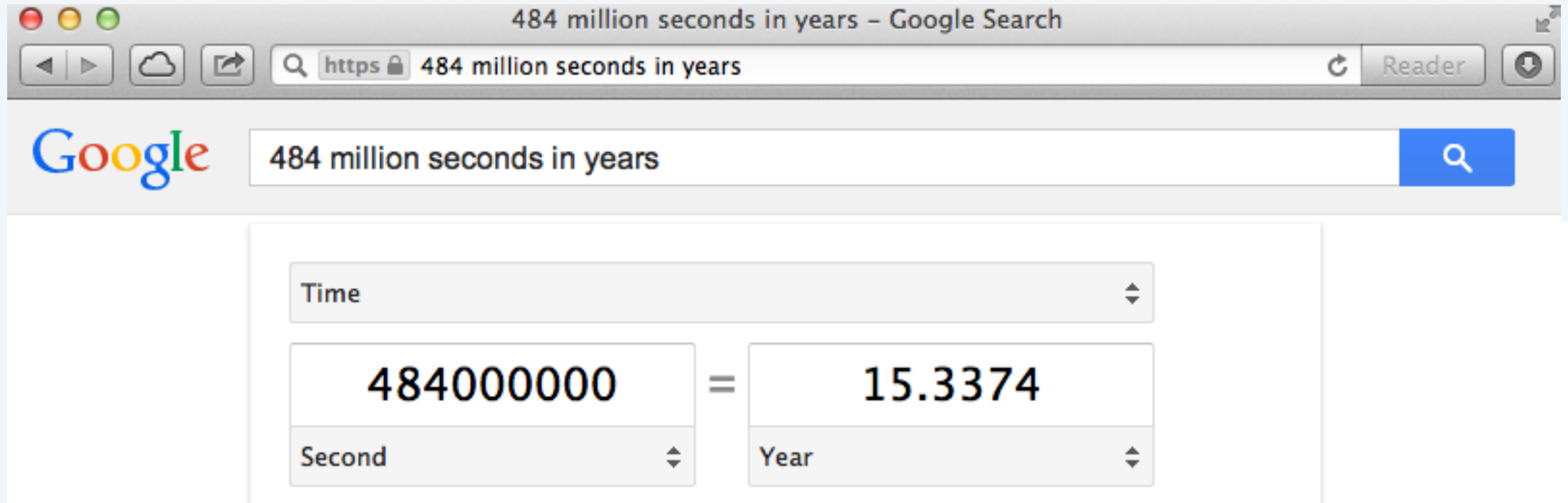
about half an hour

```
% java Generator 1000000 16000 | java ThreeSum
31903 (1985 seconds)
```

✓

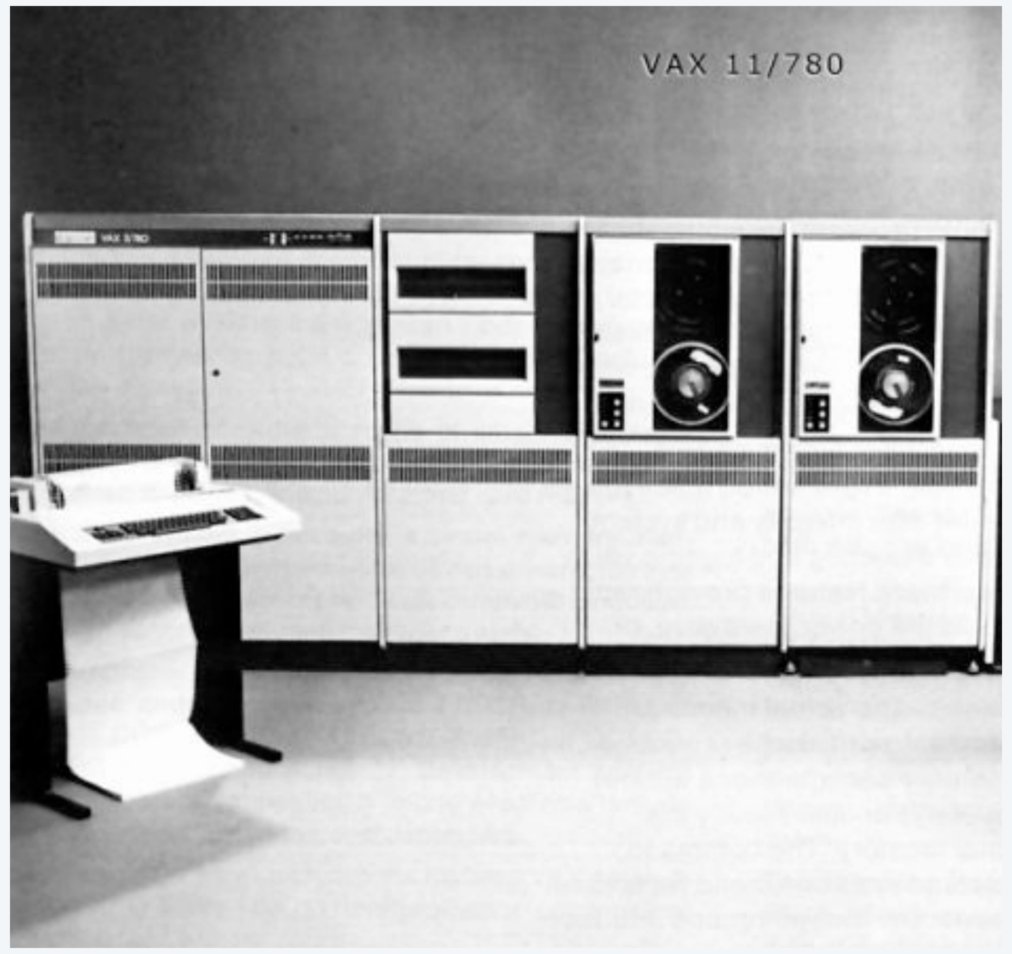
Q. How much time will this program take for $N = 1$ million?

A. 484 million seconds (more than 15 years).



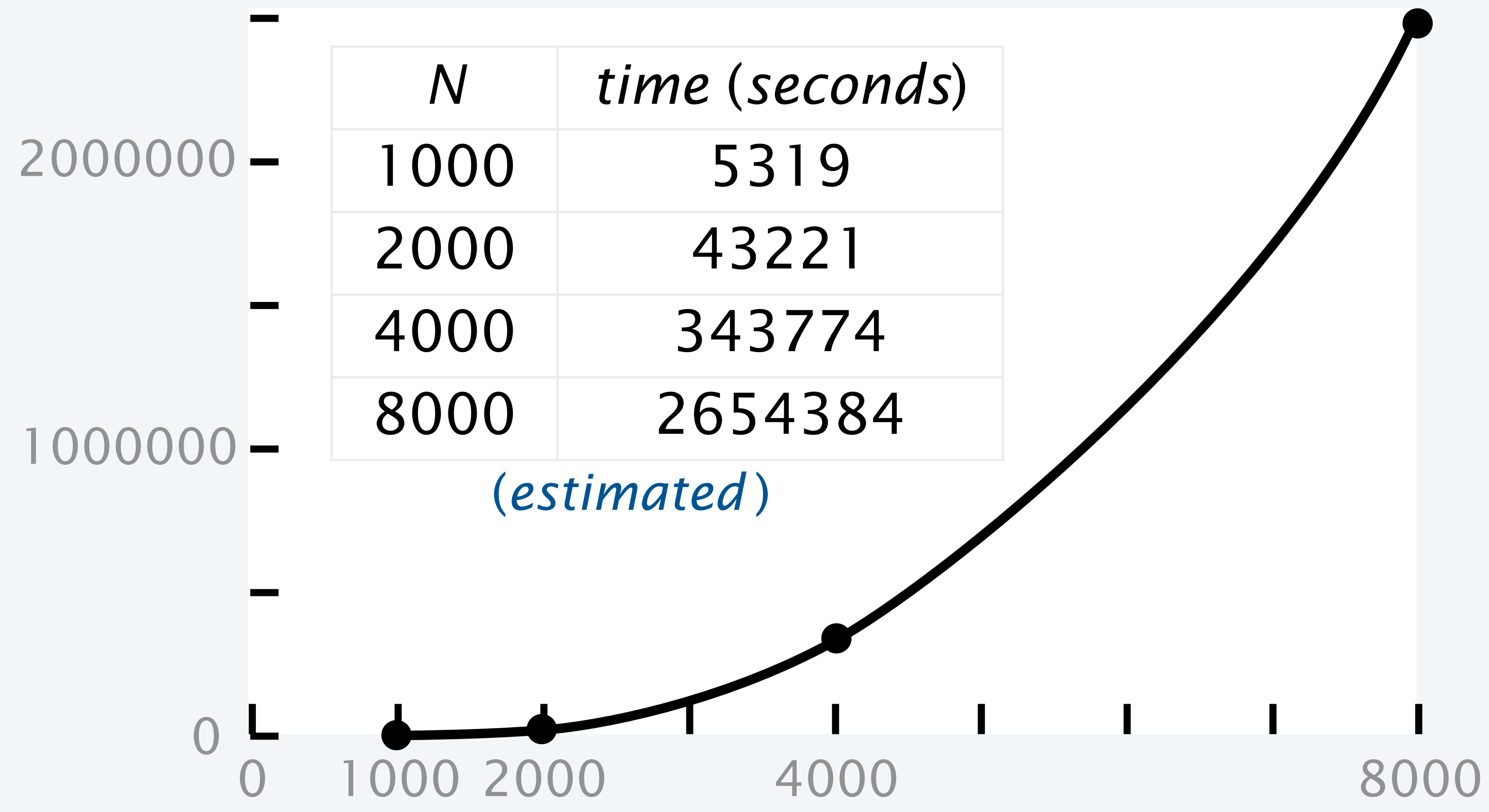
Another hypothesis

1970s



VAX 11/780

$5.2 \times 10^{-6} \times N^3$ seconds



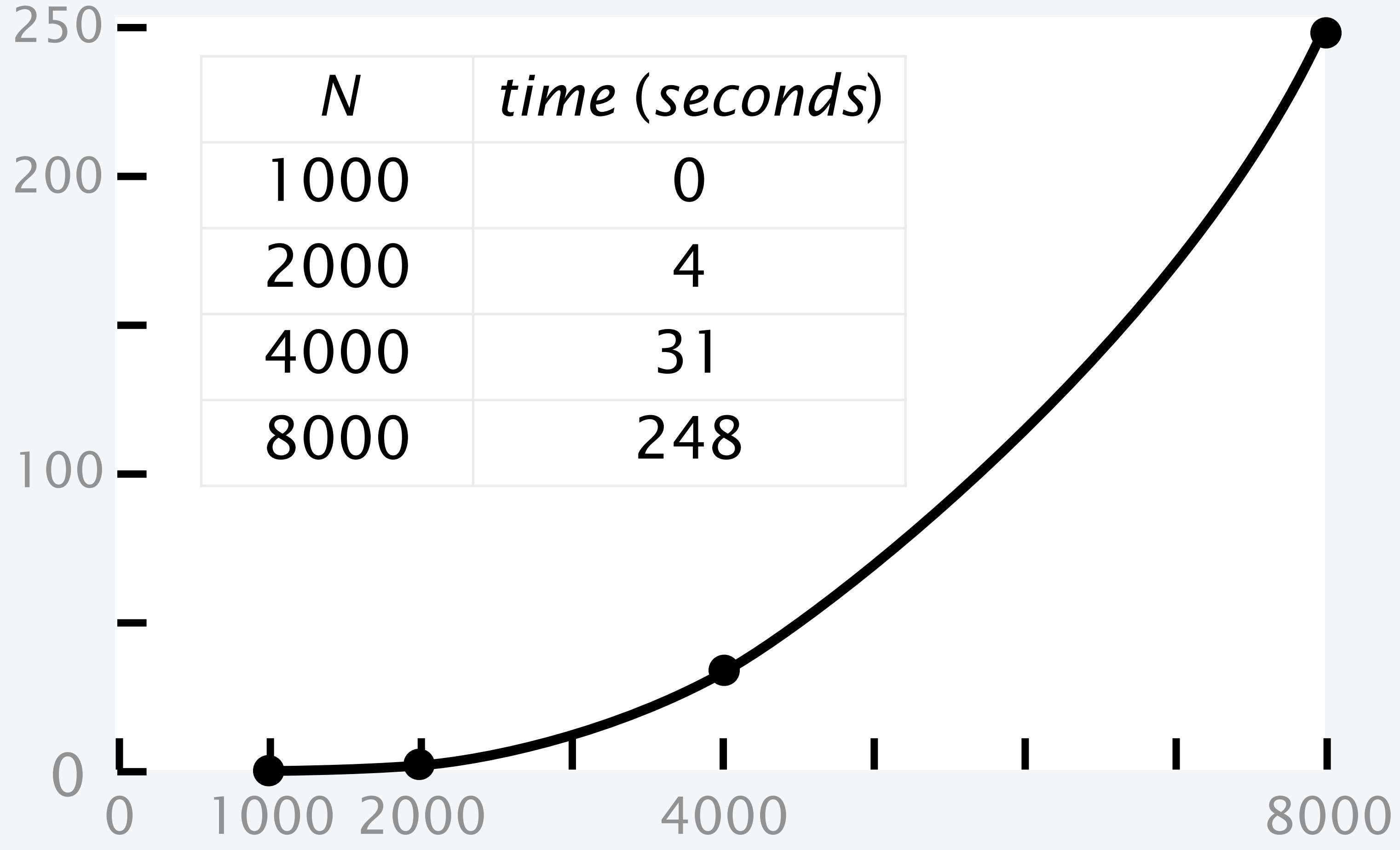
N	*time (seconds)*
1000	5319
2000	43221
4000	343774
8000	2654384

(estimated)

2010s: 10,000+ times faster

Macbook Air

$4.84 \times 10^{-10} \times N^3$ seconds



N	*time (seconds)*
1000	0
2000	4
4000	31
8000	248

Hypothesis. Running times on different computers differ by only a constant factor.

Image sources

http://commons.wikimedia.org/wiki/File:FEMA_-_2720_-_Photograph_by_FEMA_News_Photo.jpg

http://pixabay.com/en/lab-research-chemistry-test-217041/

http://upload.wikimedia.org/wikipedia/commons/2/28/Cut_rat_2.jpg

http://pixabay.com/en/view-glass-future-crystal-ball-32381/