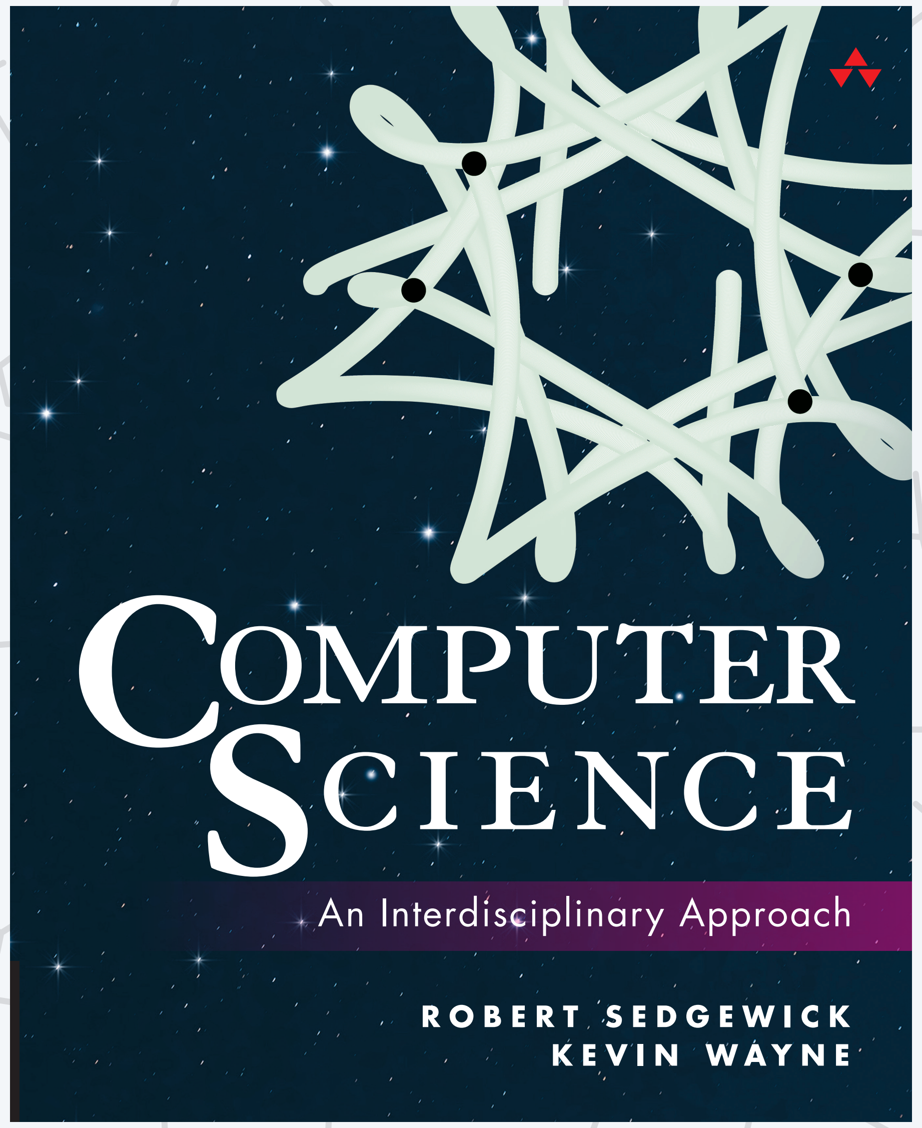
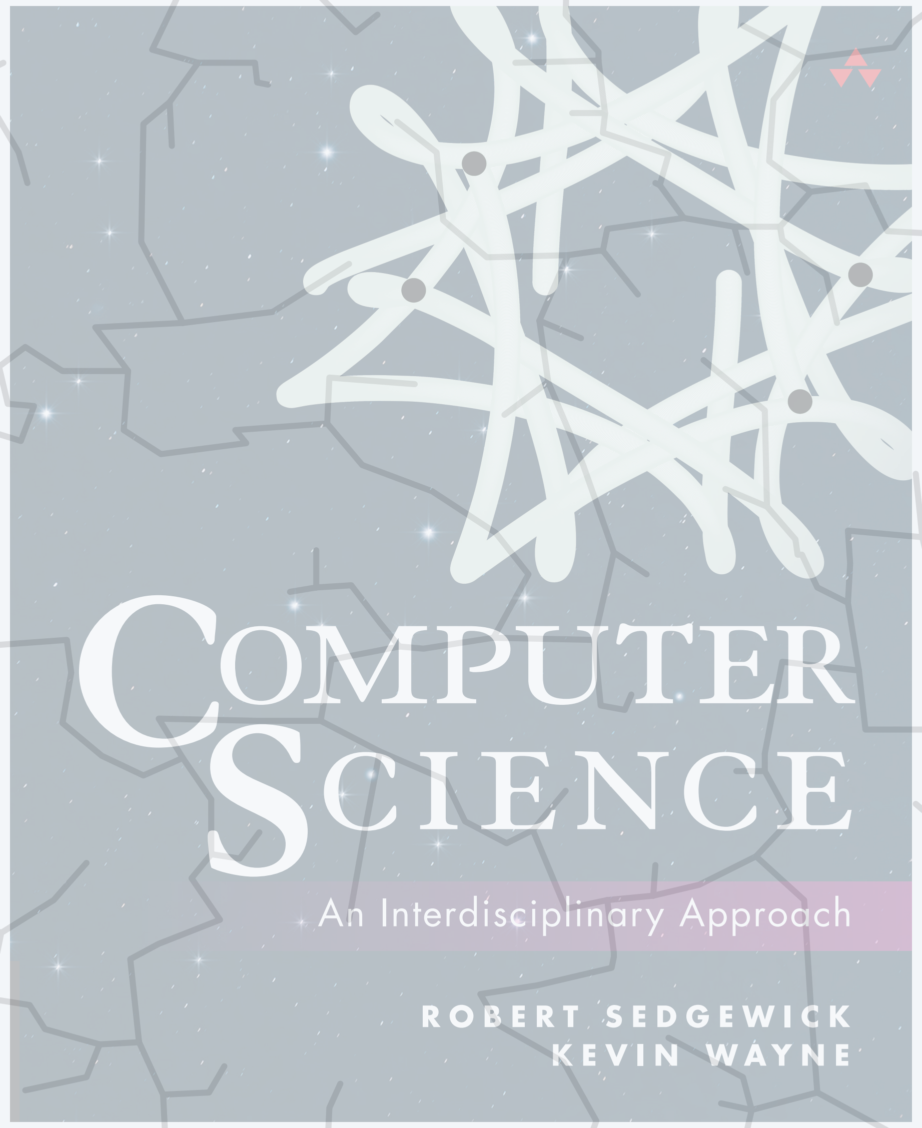


https://introcs.cs.princeton.edu

4.1 PERFORMANCE

- *the challenge*
- *empirical analysis*
- *mathematical models*
- *order-of-growth classifications*
- *memory usage*



https://introcs.cs.princeton.edu

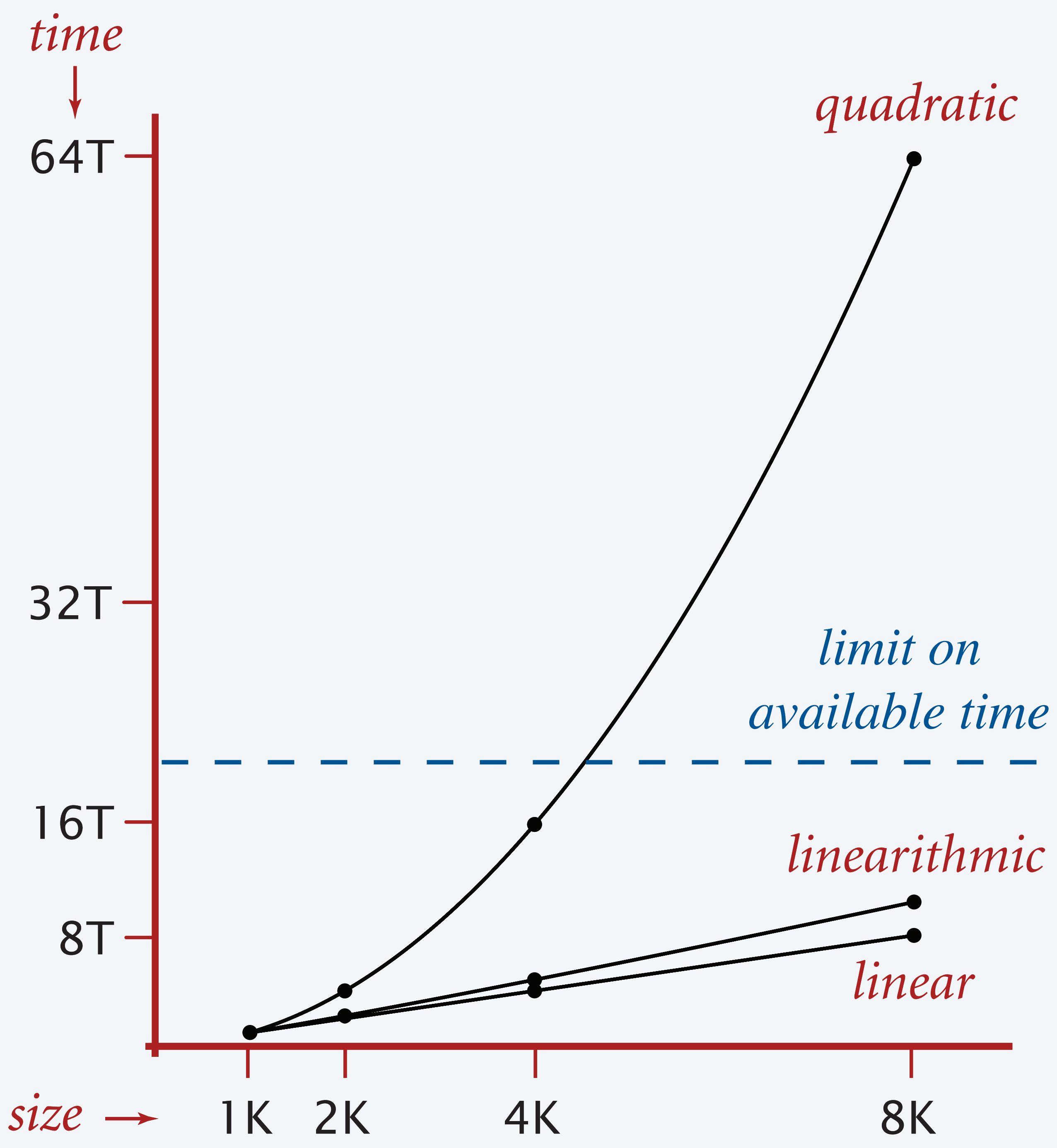
4.1 PERFORMANCE

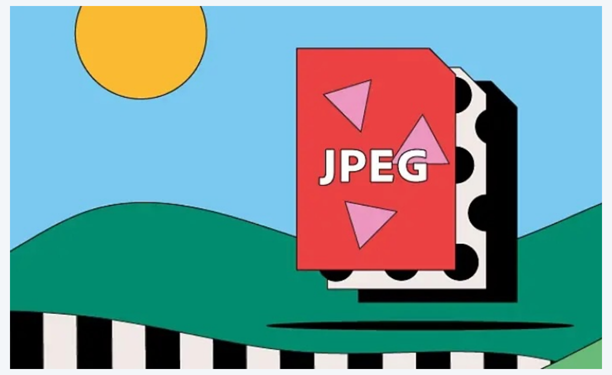
- *the challenge*
- *empirical analysis*
- *mathematical models*
- *order-of-growth classifications*
- *memory usage*

An algorithmic success story

Discrete Fourier transform.

- Multiply two univariate polynomials of degree n.
- Applications: audio processing, MRI, data compression, communications, PDEs, …
- Grade-school algorithm: $\Theta(n^2)$ steps.
- Cooley–Tukey FFT algorithm: $\Theta(n \log n)$ steps, enables new technology.

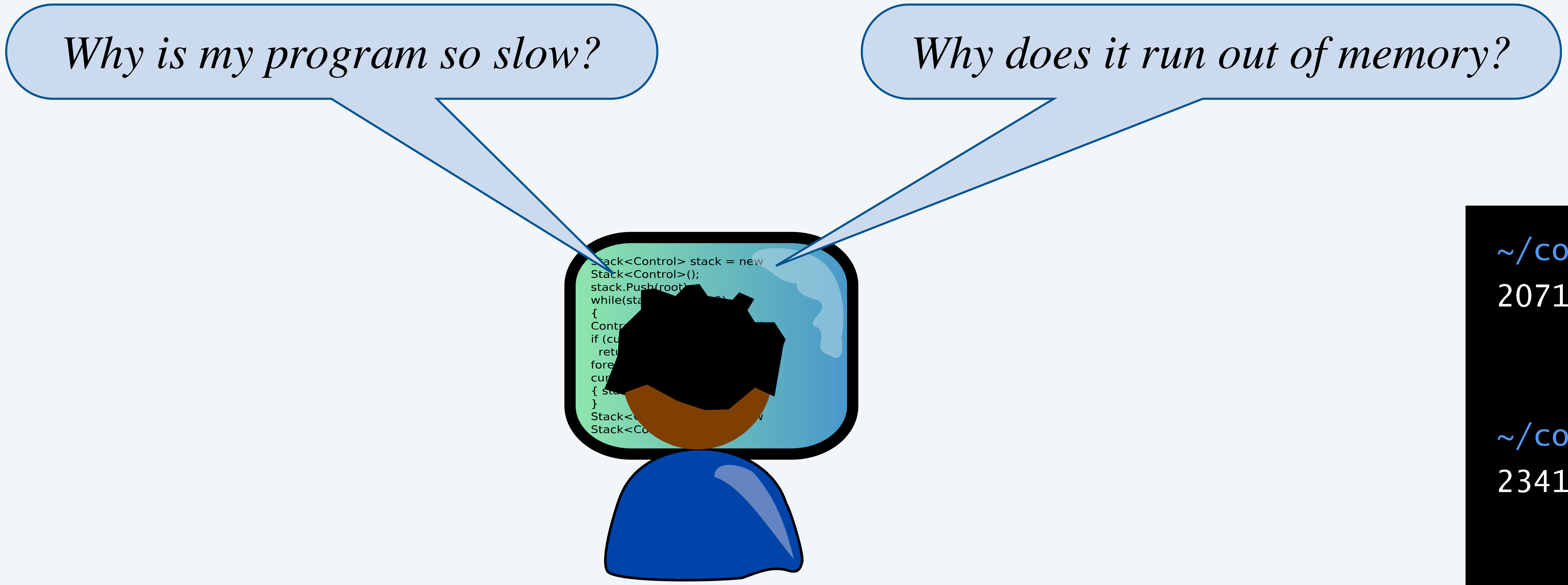




The challenge (modern version)

Q1. Will my program be able to solve a large practical input?

Q2. If not, how might I understand its performance characteristics so as to improve it?

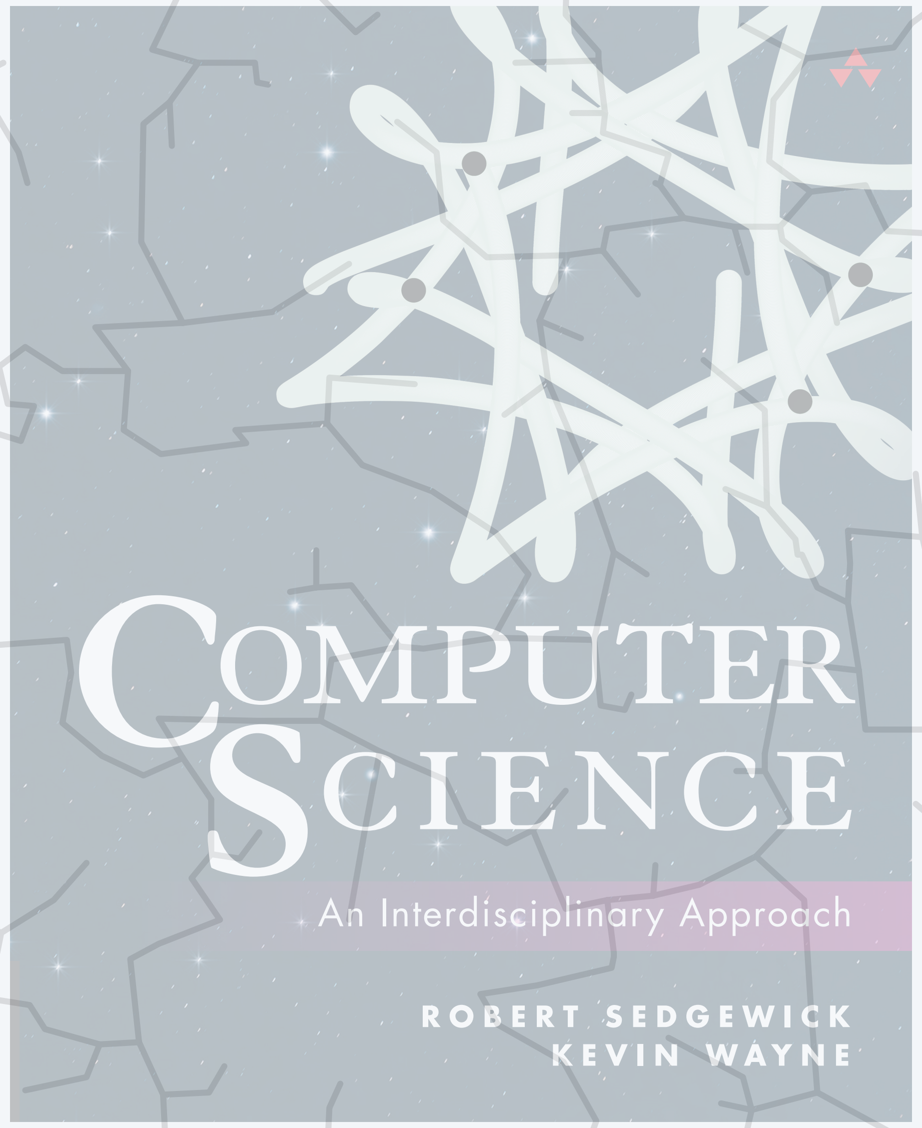



```
~/cos126/loops> java Factors 1111111111111111
2071723 536322235
```
takes a few seconds

```
~/cos126/recursion> java Fibonacci 80
23416728348467685
```
takes about 3 years (!)

```
~/cos126/loops> java Ruler 100
Exception in thread "main"
java.lang.OutOfMemoryError
```

Our approach. Combination of experiments and mathematical modeling.



https://introcs.cs.princeton.edu

4.1 Performance

- *the challenge*
- *empirical analysis*
- *mathematical models*
- *order-of-growth classifications*
- *memory usage*

Two-sum problem

Two-sum problem. Given an array with n distinct integers, how many pairs sum to zero?

	0	1	2	3	4
a[]	30	–40	20	40	–20

```
~/cos126/performance> more input5.txt
30 -40 20 40 -20

~/cos126/performance> java-introcs TwoSum < input5.txt
2

~/cos126/performance> more input1M.txt
30 -40 20 40 -20 ...

~/cos126/performance> java-introcs TwoSum < input1M.txt
...
```

can my program solve large instances?

i	j	a[i]	a[j]	sum
1	3	–40	40	0
2	4	20	–20	0

Two-sum implementation

Two-sum problem. Given an array with n distinct integers, how many pairs sum to zero?

Brute-force algorithm.

- Process all distinct pairs.
- Increment counter when pair sums to 0.

a[]	0	1	2	3	4
	30	–40	20	40	–20

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

avoid double counting pairs (e.g., 1–3 and 3–1)

i	j	a[i]	a[j]	sum	
0	1	30	–40	–10	
0	2	30	20	50	
0	3	30	40	70	
0	4	30	–20	10	
1	2	–40	20	–20	
1	3	–40	40	0	✔
1	4	–40	–20	–60	
2	3	20	40	60	
2	4	20	–20	0	✔
3	4	40	–20	20	

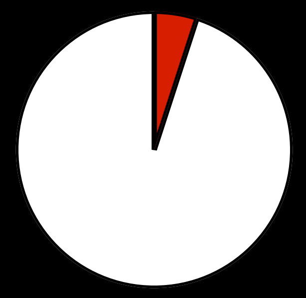
Q. How long will this program take for $n = 1$ million integers?

Measuring the running time

Running time. Run the program for inputs of varying size; measure running time.

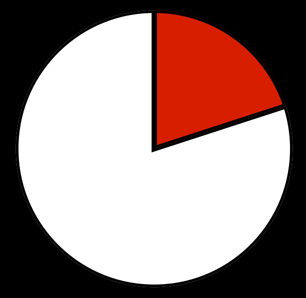
Observation. The running time $T(n)$ increases as a function of the input size n.

```
~/cos126/performance> java-introcs TwoSum < 10Kints.txt
12
```



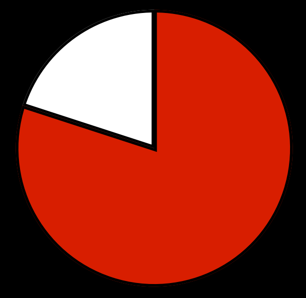
tick tick

```
~/cos126/performance> java-introcs TwoSum < 25Kints.txt
14
```



tick tick tick tick tick tick
tick tick tick tick tick tick

```
~/cos126/performance> java-introcs TwoSum < 50Kints.txt
35
```



tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick

Measuring the running time

Running time. Run the program for inputs of varying size; measure running time.

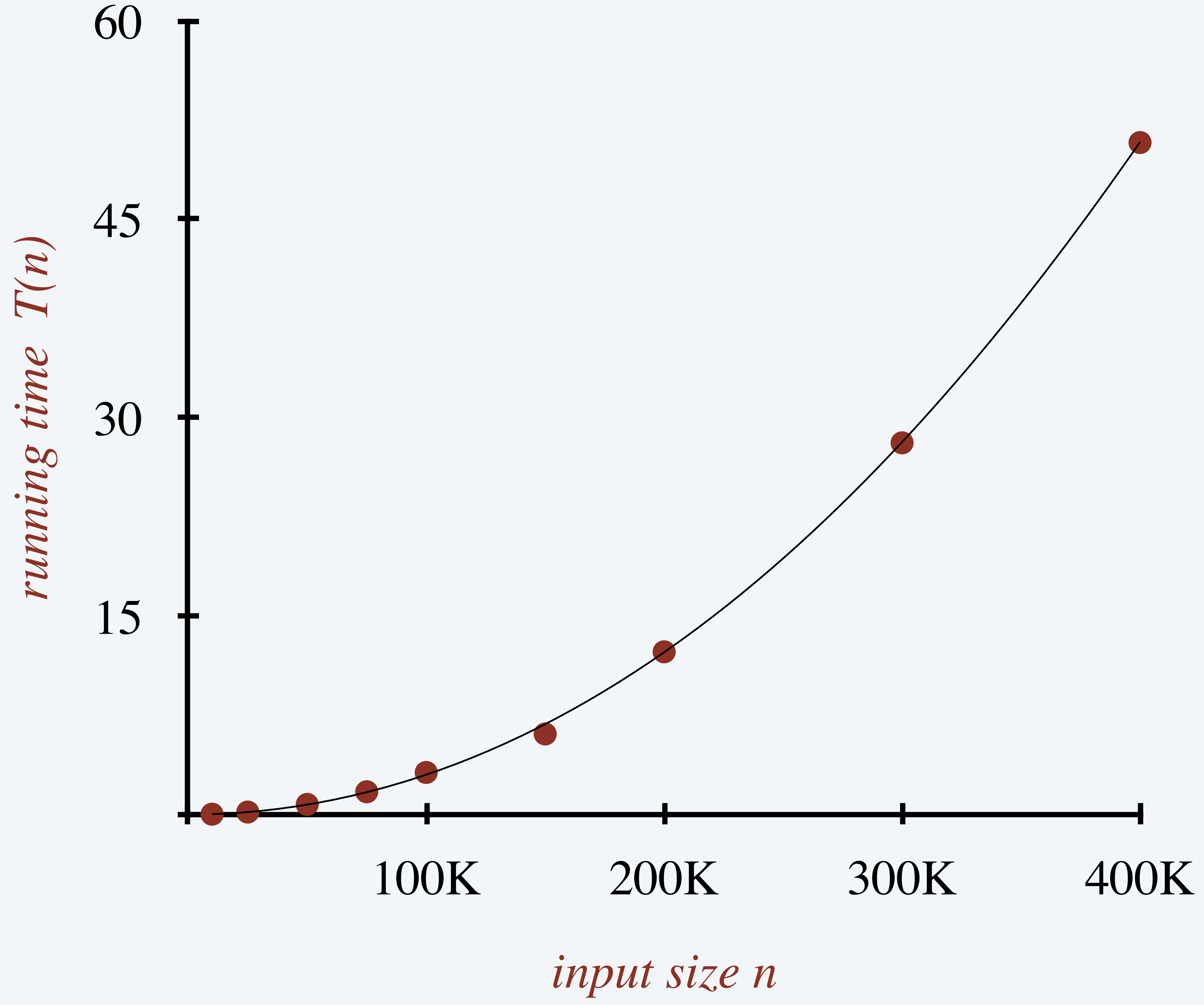
n	time (seconds) †
10,000	0.025
25,000	0.187
50,000	0.766
75,000	1.72
100,000	3.18
150,000	6.09
200,000	12.3
300,000	28.1
400,000	50.8

† *Apple M2 Pro with 32 GB memory running OpenJDK 11 on MacOS Ventura*

Data analysis: standard plot

Standard plot. Plot running time $T(n)$ vs. input size n.

n	time (seconds) †
10,000	0.025
25,000	0.187
50,000	0.766
75,000	1.72
100,000	3.18
150,000	6.09
200,000	12.3
300,000	28.1
400,000	50.8



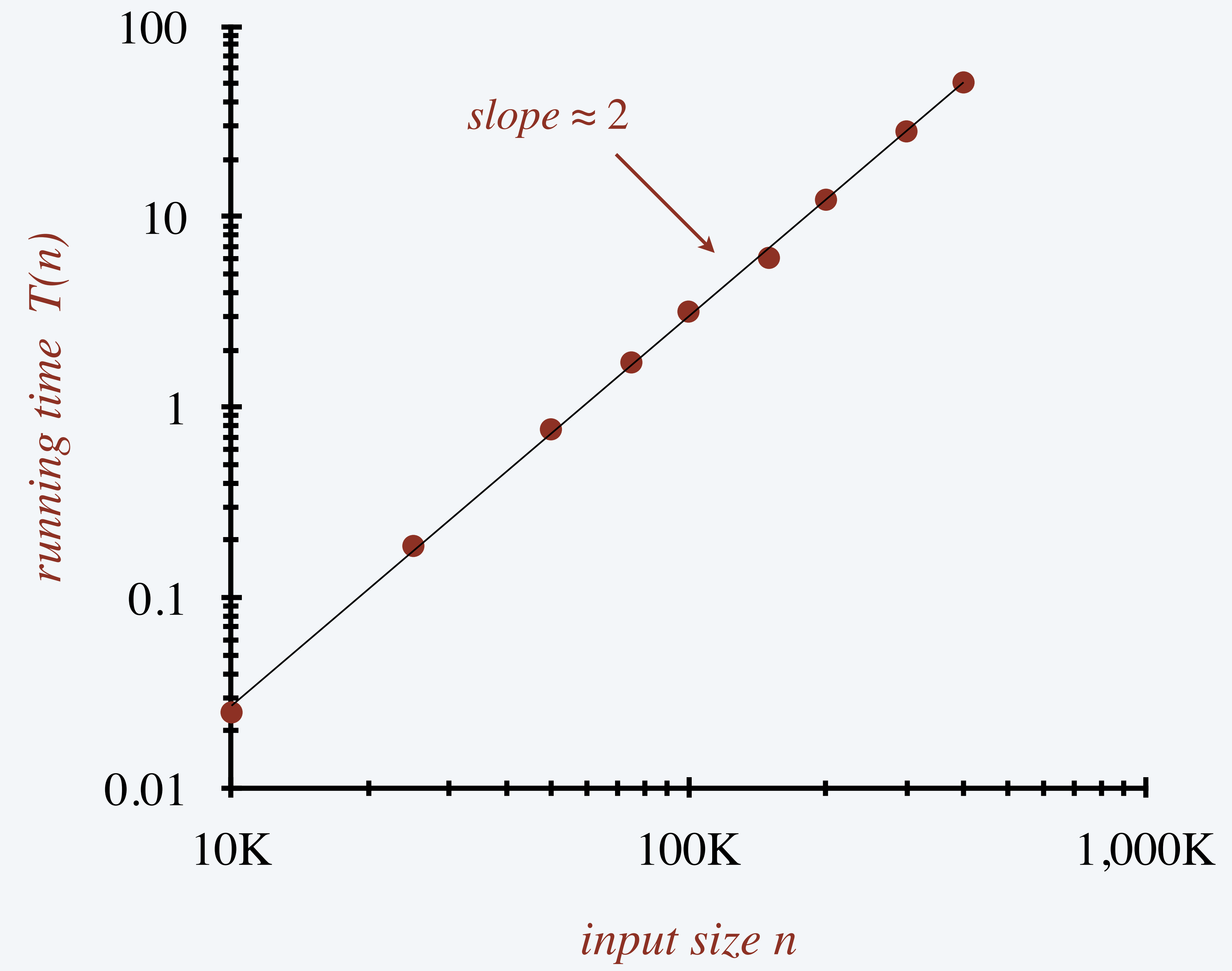
Hypothesis. The running time obeys a power law: $T(n) = a \times n^b$ seconds.

Questions. How to validate hypothesis? How to estimate constants a and b ?

Data analysis: log-log plot

Log-log plot. Plot running time $T(n)$ vs. input size n using log-log scale.

n	time (seconds) †
10,000	0.025
25,000	0.187
50,000	0.766
75,000	1.72
100,000	3.18
150,000	6.09
200,000	12.3
300,000	28.1
400,000	50.8



Regression. Fit straight line through data points.

"quadratic algorithm" (stay tuned)

Hypothesis. The running time $T(n)$ is about $3.18 \times 10^{-10} \times n^2$ seconds.

Doubling test: estimating the exponent b

Doubling test. Run program, doubling the size of the input.

- Assume running time obeys a power law $T(n) = a \times n^b$.
- Estimate $b = \log_2$ ratio.

n	time (seconds) †	ratio	$\log_2$ ratio
10,000	0.025		–
20,000	0.15	6.0	2.6
40,000	0.55	3.7	1.9
80,000	2.0	3.6	1.9
160,000	8.1	4.1	2.0
320,000	32.5	4.0	2.0

$\log_2 (8.1 / 2.0) = 2.02$

seems to converge to a constant $b \approx 2.0$

$$\frac{T(n)}{T(n/2)} = \frac{an^b}{a(n/2)^b} = 2^b$$

$$\Longrightarrow \quad b = \log_2 \frac{T(n)}{T(n/2)}$$

why the $\log_2$ ratio works

Doubling test: estimating the leading coefficient a

Doubling test. Run program, doubling the size of the input.

- Assume running time obeys a power law $T(n) = a \times n^b$.
- Estimate $b = \log_2$ ratio.
- Estimate a by solving $T(n) = a \times n^b$ for a sufficiently large value of n.

n	time (seconds) †	ratio	$\log_2$ ratio
10,000	0.025		–
20,000	0.15	6.0	2.6
40,000	0.55	3.7	1.9
80,000	2.0	3.6	1.9
160,000	8.1	4.1	2.0
320,000	32.5	4.0	2.0

$$32.5 = a \times 320{,}000^2$$

$$\Rightarrow \quad a = 3.17 \times 10^{-10}$$

Hypothesis. Running time is about $3.17 \times 10^{-10} \times n^2$ seconds. ← *almost identical hypothesis to one obtained via regression (but less work)*

Analysis of algorithms: quiz 1

Estimate the running time to solve a problem of size $n = 64{,}000$.

- **A.** 400 *seconds*
- **B.** 600 *seconds*
- **C.** 800 *seconds*
- **D.** 1,600 *seconds*

n	time (seconds)
2,000	0.08
4,000	0.40
8,000	3.20
16,000	26.0
32,000	205.0
64,000	?

Machine invariance

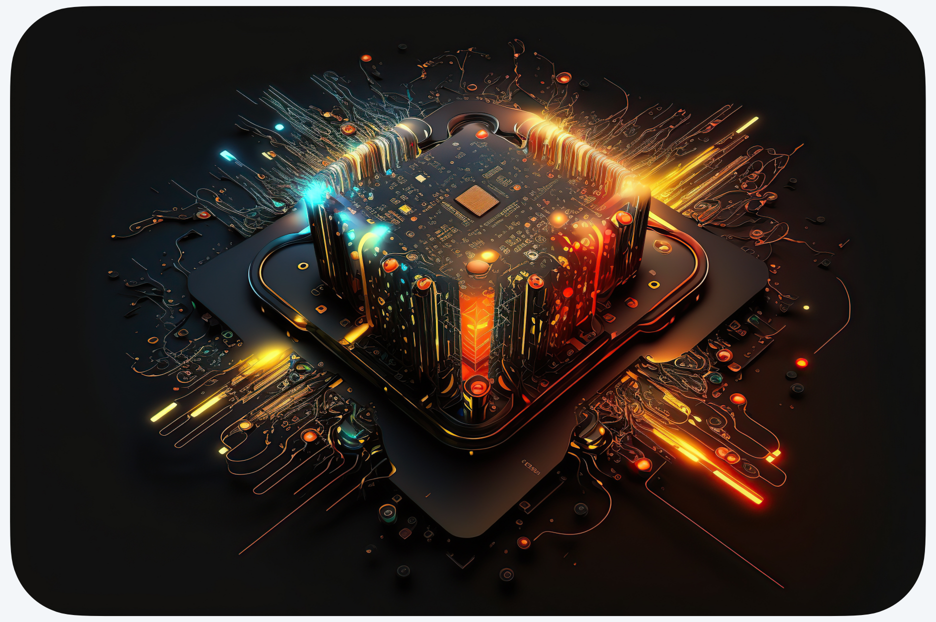
Hypothesis. Running times on different computers differ by (roughly) a constant factor.

Note. That factor can be several orders of magnitude.

1970s
(VAX-11/780)

10,000× *faster*

2020s
(Macbook Pro M2)



futuristic counterexample?
(quantum computer)

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent b in power law $T(n) = a \times n^b$

System dependent effects.

- Hardware: CPU, memory, cache, …
- Software: compiler, interpreter, garbage collector, …
- System: operating system, network, other apps, …

determines leading coefficient a in power law $T(n) = a \times n^b$

M2

OpenJDK

22.04 LTS

Bad news. Sometimes difficult to get accurate measurements.

Context: the scientific method

Experimental algorithmics is an example of the scientific method.

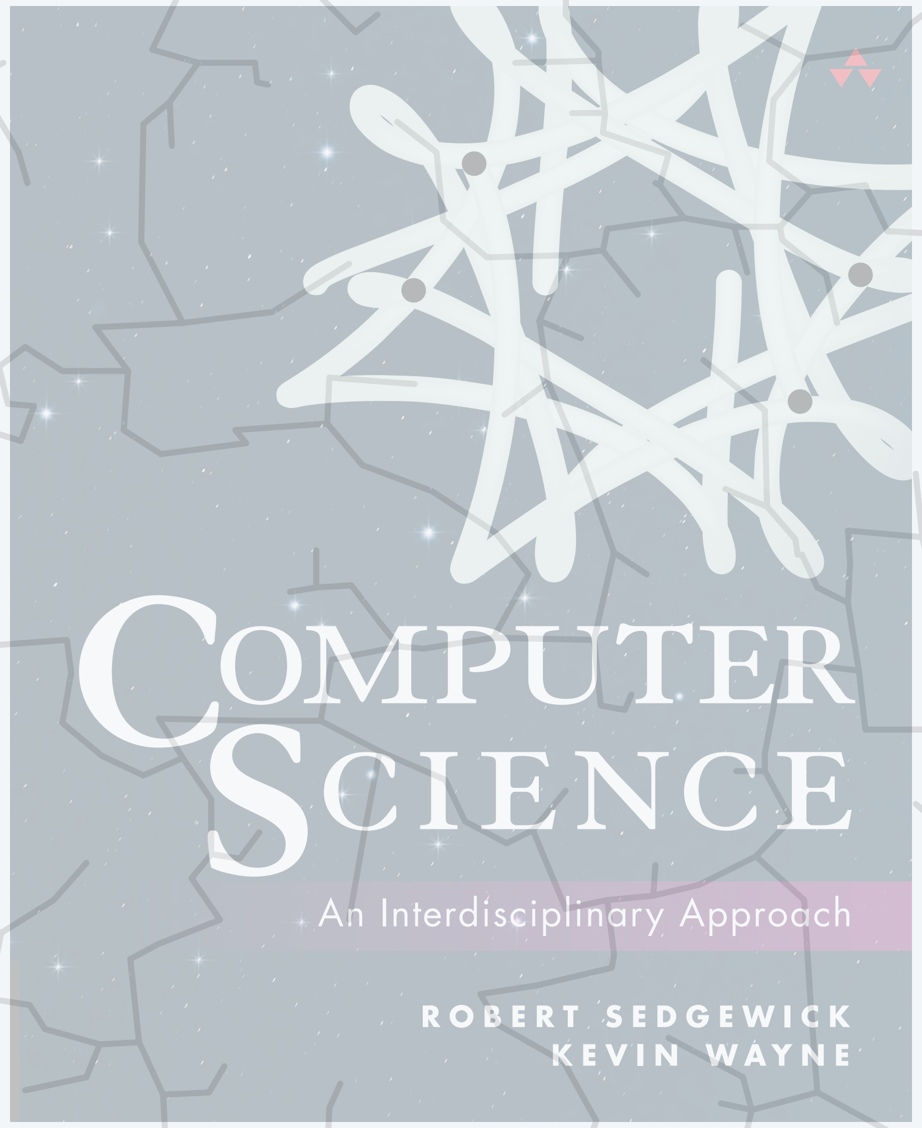
Chemistry
(1 experiment)

Biology
(1 experiment)

Computer Science
(1 million experiments)

Physics
(1 experiment)

Good news. Experiments are easier and cheaper than other sciences.



https://introcs.cs.princeton.edu

4.1 Performance

- *the challenge*
- *empirical analysis*
- *mathematical models*
- *order-of-growth classifications*
- *memory usage*

Mathematical models for running time

Total running time: sum of frequency × cost for all operations.

- Frequency depends on algorithm and input data.
- Cost depends on CPU, compiler, operating system, …

Example: one-sum

Q. How many operations as a function of input size n ?

```
int count = 0;
for (int i = 0; i < n; i++)
   if (a[i] == 0)
      count++;
```

operation	cost (ns) †	frequency
variable declaration	2/5	2
assignment statement	1/5	2
less than compare	1/5	$n + 1$
equal to compare	1/10	n
array access	1/10	n
increment	1/10	n to $2n$

tedious to count exactly

† *representative estimates (with some poetic license)*

Simplification 1: cost model

Cost model. Use some elementary operation as a proxy for running time. ← *array accesses, compares, API calls, floating-point operations, …*

```
int count = 0;
for (int i = 0; i < n; i++)
   if (a[i] == 0)
      count++;
```

← *exactly n array accesses*

operation	cost (ns) †	frequency
variable declaration	2/5	2
assignment statement	1/5	2
less than compare	1/5	$n + 1$
equal to compare	1/10	n
array access	1/10	n
increment	1/10	n to $2n$

← *cost model = array accesses*

Simplification 2: asymptotic notations

Tilde notation. Discard lower-order terms.

Big Theta notation. Discard lower-order terms and leading coefficient.

formal definitions involve limits

function	tilde notation	big Theta
$4\,n^5 + 20\,n + 16$	$\sim 4\,n^5$	$\Theta(n^5)$
$7\,n^2 + 10\,n \log_2 n + 56$	$\sim 7\,n^2$	$\Theta(n^2)$
$\frac{1}{6}\,n^3 - \frac{1}{2}\,n^2 + \frac{1}{3}\,n$	$\sim \frac{1}{6}\,n^3$	$\Theta(n^3)$

discard lower-order terms
(e.g., n = 1,000: 166.67 million vs. 166.17 million)

"order of growth"

Rationale.

- When n is large, lower-order terms are negligible.
- When n is small, we don't care.

Example: two-sum analysis

Goal. Estimate running time as a function of input size n.

```
int count = 0;
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      if (a[i] + a[j] == 0)
         count++;
```

when $i = 1$

$$(n-1) + (n-2) + \ldots + 2 + 1 + 0 = \frac{n(n-1)}{2} = \binom{n}{2}$$

when $i = 0$

Step 1. Use array accesses as cost model.

Step 2. $\Theta(n^2)$ array accesses.

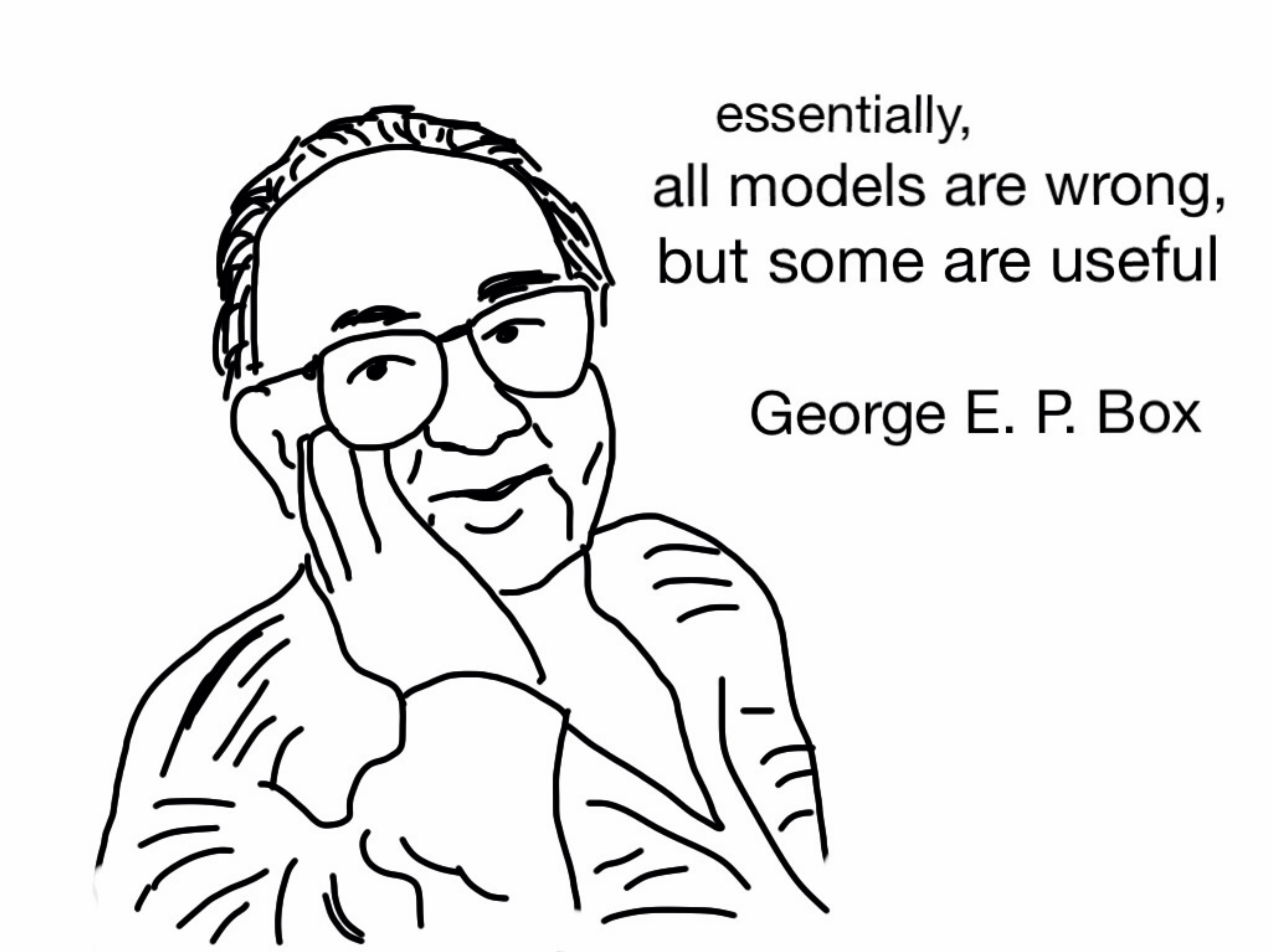
Bottom line. Mathematical model explains and supports empirical experiments.

provides exponent in power law (but not leading coefficient)

All models are wrong

Model deficiencies.

- Input size n does not go to infinity. ← *computers (and the universe) are finite*
- Can be inaccurate when n is small.
- Cost model may not be a perfect proxy for running time.
- …



Estimate running time as a function of n ?

A. $\Theta(n)$

B. $\Theta(n^2)$

C. $\Theta(n^3)$

D. $\Theta(n^4)$

```
int count = 0;
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
         if (a[i] + a[j] >= a[k])
            count++;
         }
      }
   }
}
```

Analysis of algorithms: quiz 3

Estimate running time as a function of n ?

A. $\Theta(n)$

B. $\Theta(n^2)$

C. $\Theta(n^3)$

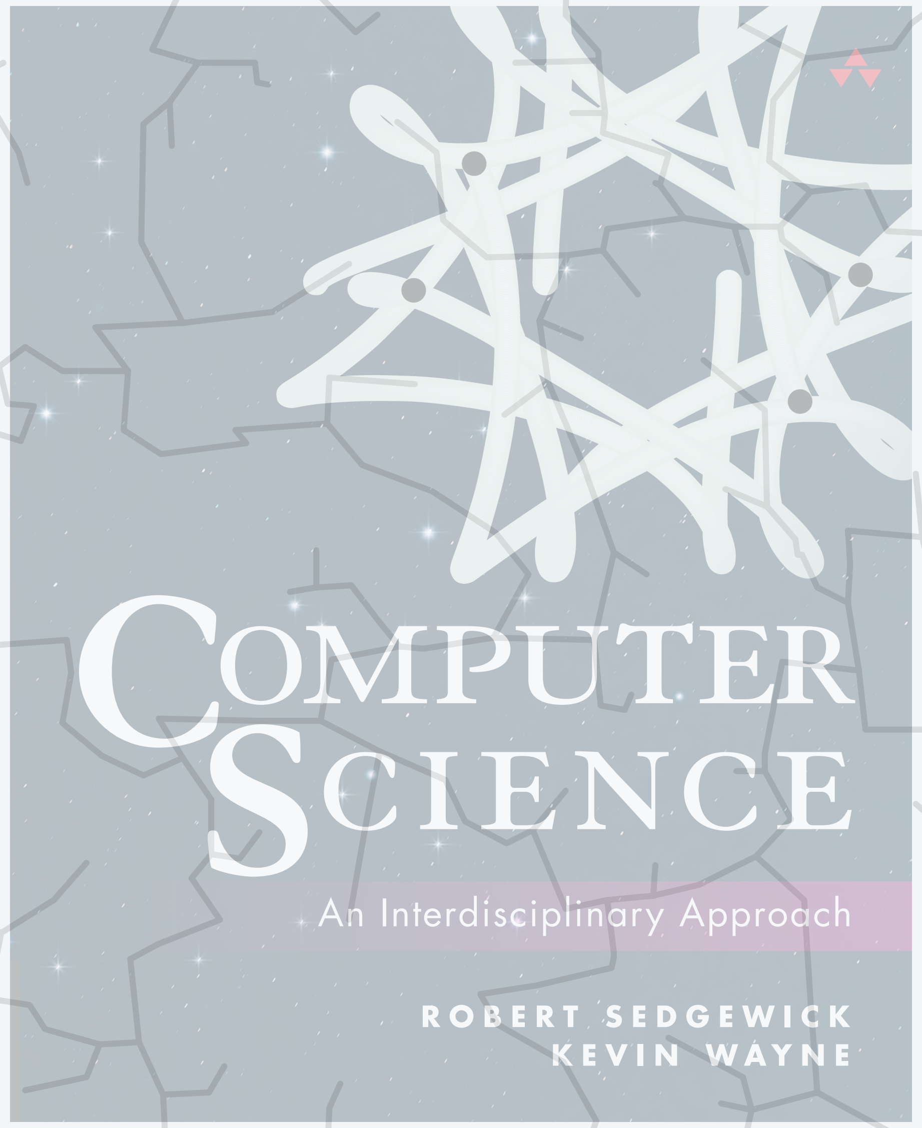
D. $\Theta(n^4)$

```
int count = 0;
for (int i = 0; i < n; i++) {

   for (int j = 0; j < n; j++) {
      if (a[i] == 0)
         count++;
   }

   for (int k = 0; k < n; k++) {
      if (a[i] >= a[k])
         count++;
   }

}
```



https://introcs.cs.princeton.edu

4.1 PERFORMANCE

- the challenge
- empirical analysis
- mathematical models
- order-of-growth classifications
- memory usage

Key questions and answers

Q. Does the running time of my program approximately obey a power law ?

A. Probably yes. Might also have a $\log n$ factor.

Q. How do you know?

A1. Computer scientists have observed power laws for many many specific algorithms.

A2. Program built from simple constructors (statements, loops, nesting, function calls).

Ex. Logarithmic running time.

```
int count = 0;
for (int i = 1; i <= n; i = i*2)
   count++;
```

code fragment takes $\Theta(\log n)$ time

Common order-of-growth classifications

order of growth	emoji	name	typical code framework	description	example	$T(2n) / T(n)$
$\Theta(1)$	😍	**constant**	`a = b + c;`	statement	*add two numbers*	1
$\Theta(\log n)$	😎	**logarithmic**	`for (int i = n; i >= 1; i /= 2)` `{ ... }`	divide in half	*binary search*	~ 1
$\Theta(n)$	😁	**linear**	`for (int i = 0; i < n; i++)` `{ ... }`	single loop	*find the maximum*	2
$\Theta(n \log n)$	😀	**linearithmic**	*mergesort* *(stay tuned)*	divide and conquer	*mergesort*	~ 2

$\Theta(n^2)$	😕	**quadratic**	`for (int i = 0; i < n; i++)`
			`for (int j = 0; j < n; j++)`
			`{ ... }`
				double loop	*check all pairs*	4

$\Theta(n^3)$	☹️	**cubic**	`for (int i = 0; i < n; i++)`
			`for (int j = 0; j < n; j++)`
			`for (int k = 0; k < n; k++)`
			`{ ... }`
				triple loop	*check all triples*	8

$\Theta(2^n)$	👿	**exponential**	*towers of Hanoi*	exhaustive search	*check all subsets*	2^n

Examples of order-of-growth

computation	implementation	order of growth

dot product	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`   sum += a[i] * b[i];`
		$\Theta(n)$

matrix addition	`for (int i = 0; i < n; i++)`
	`   for (int j = 0; j < n; j++)`
	`      c[i][j] = a[i][j] + b[i][j];`
		$\Theta(n^2)$

matrix multiplication	`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];`
		$\Theta(n^3)$

ruler function	`public static int ruler(int n) {`
	`   if (n == 0) return " ";`
	`   return ruler(n-1) + n + ruler(n-1);`
	`}`
		$\Theta(2^n)$

note: input size is n^2, not n

Analysis of algorithms: quiz 4

What is order of growth of the running time as a function of n ?

Hint: use array accesses as cost model.

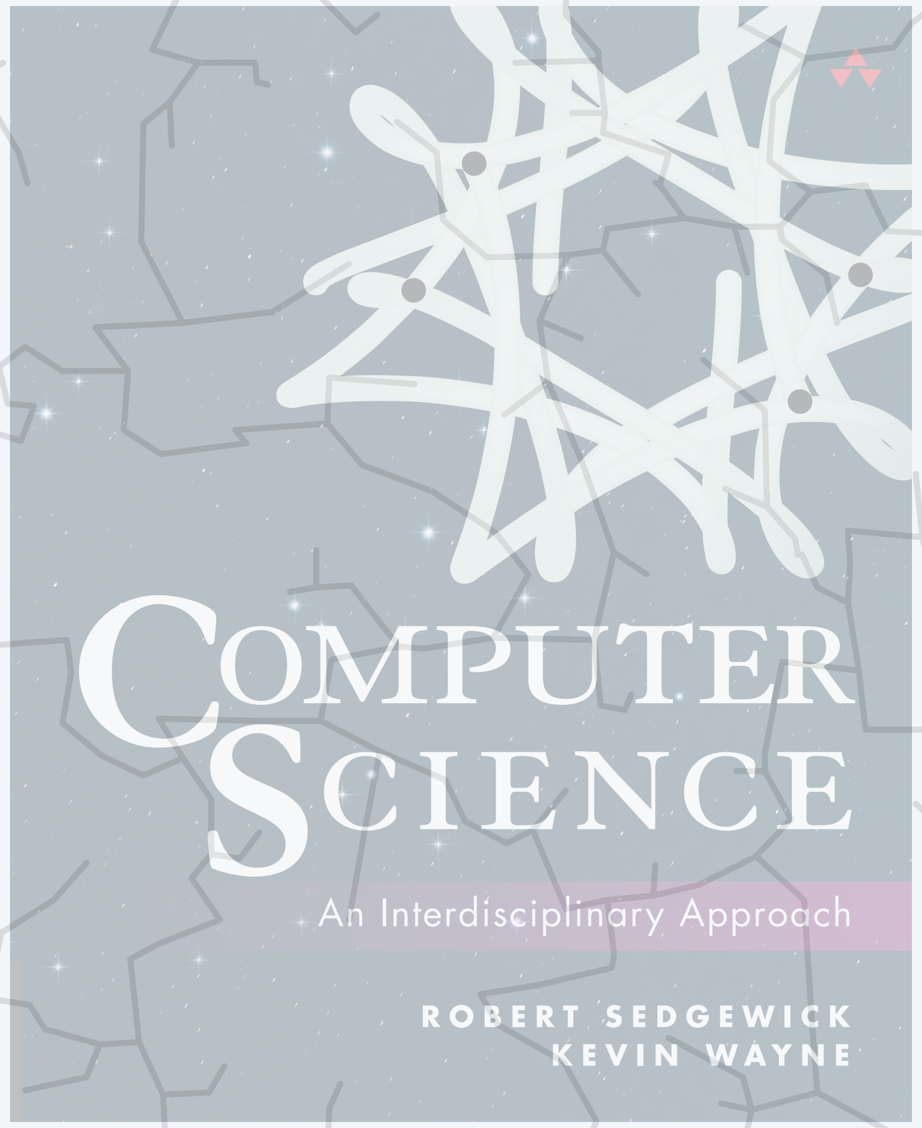
```
int count = 0;
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      for (int k = n; k >= 1; k = k/2)
         if (a[i] + a[j] >= a[k-1])
            count++;
```

A. $\Theta(n^2)$

B. $\Theta(n^2 \log n)$

C. $\Theta(n^3)$

D. $\Theta(n^3 \log n)$



https://introcs.cs.princeton.edu

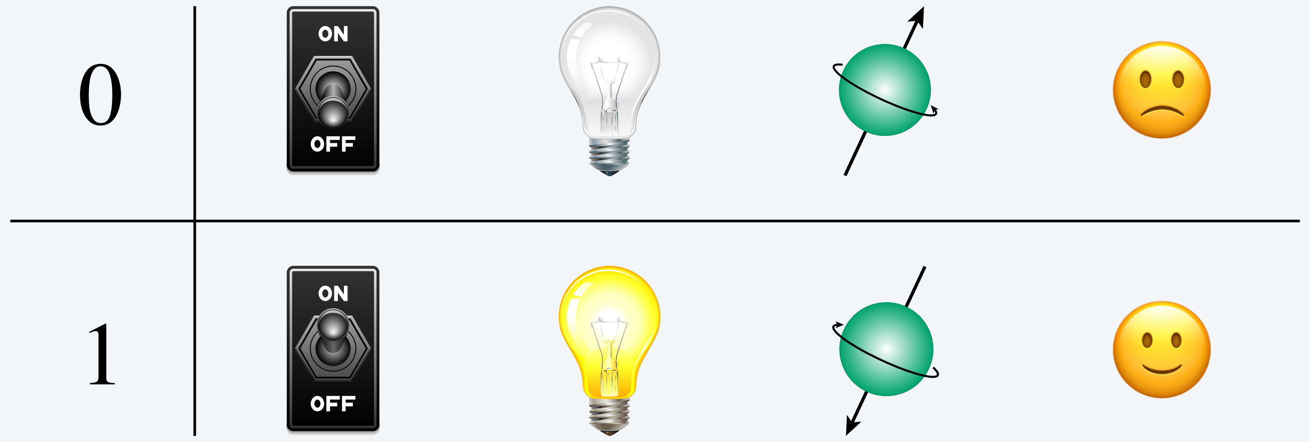
4.1 Performance

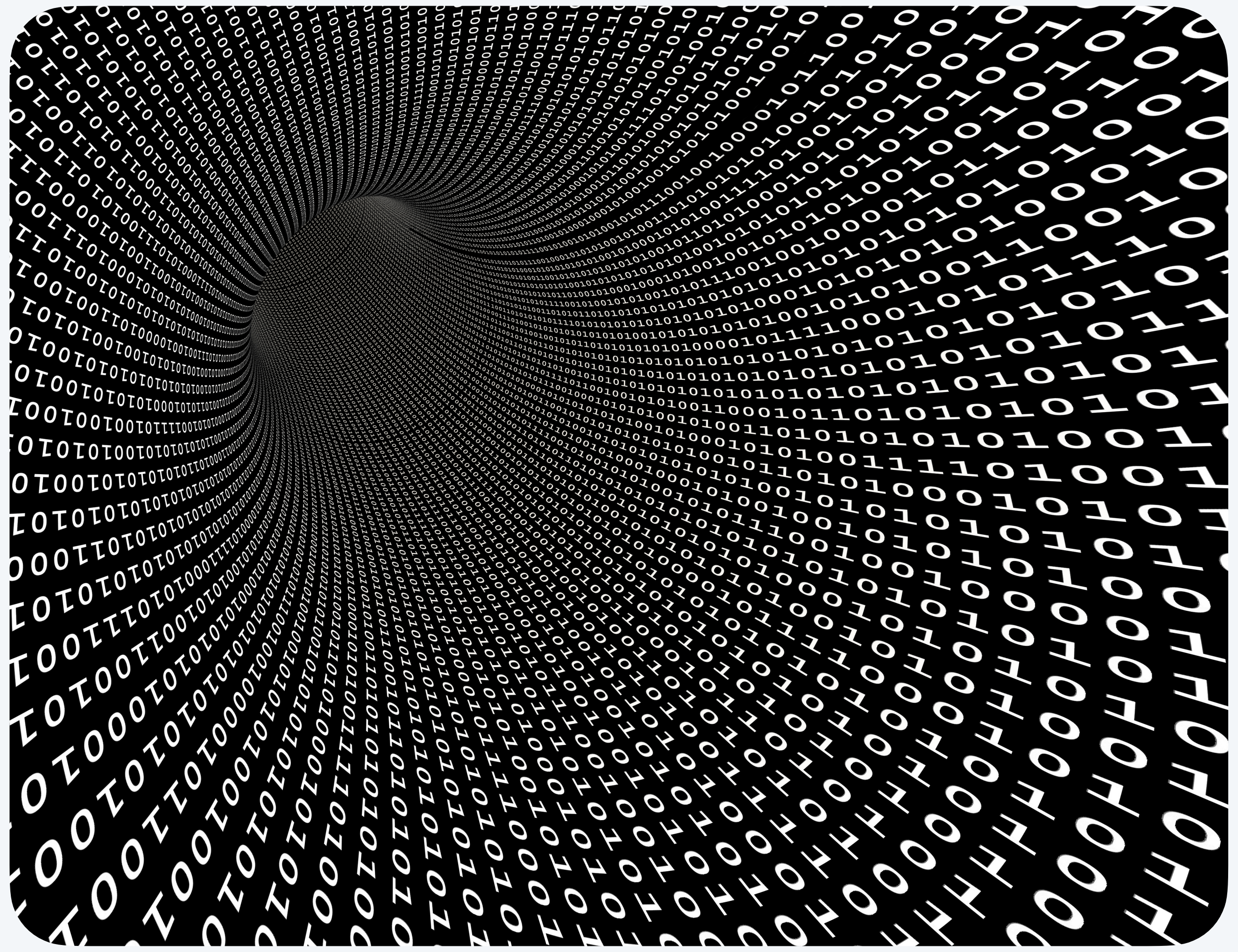
- the challenge
- empirical analysis
- mathematical models
- order-of-growth classifications
- memory usage

Memory basics

Bit (binary digit). 0 or 1.

Byte (8 bits). Smallest addressable unit of computer memory.





term	symbol	quantity
byte	B	8 bits
kilobyte	KB	1000 bytes
megabyte	MB	1000^2 bytes
gigabyte	GB	1000^3 bytes
terabyte	TB	1000^4 bytes

some define using powers of 2
($MB = 2^{10}$ *bytes*)

6 GB main memory,
1 TB internal storage

Typical memory usage in Java for primitive types and arrays

type	bytes	
`boolean`	1	
`byte`	1	
`char`	2	
`int`	4	← *32 bits*
`float`	4	
`long`	8	
`double`	8	← *64 bits*
`String`	$n + 40$	← *ASCII string of length n*

built-in types

type	bytes	
`boolean[]`	$1n + 24$	
`int[]`	$4n + 24$	
`double[]`	$8n + 24$	← *array overhead = 24 bytes*

one-dimensional arrays (length n)

type	bytes
`boolean[][]`	$\sim 1\, n^2$
`int[][]`	$\sim 4\, n^2$
`double[][]`	$\sim 8\, n^2$

two-dimensional arrays (n-by-n)

Analysis of algorithms: quiz 5

How much memory (in bytes) does *result* use as a function of n?

A. $\sim 2n$ bytes

B. $\sim n^2$ bytes

C. $\sim 2n^2$ bytes

D. $\sim 2^n$ bytes

```
public class Mystery {

   public static String f(int n) {
      if (n == 0) return "";
      return f(n-1) + "*" + f(n-1);
   }

   public static void main(String[] args) {
      int n = Integer.parseInt(args[0]);
      String result = f(n);
      StdOut.println(result);
   }
}
```

Turning the crank: summary

Running time analysis. Analyze running time $T(n)$ as a function of input size n.

Empirical analysis.

- Run code on specific machine and inputs and measure running times.
- Formulate a hypothesis for running time.
- Enables us to make predictions.

Mathematical analysis.

- Analyze algorithm on abstract machine.
- Count frequency of dominant operations. ← *use big-Theta notation to simplify analysis*
- Enables us to explain behavior.

This course. Learn to use both.

Credits

media	source	license
Charles Babbage	The Illustrated London News	public domain
Babbage's Analytical Engine	xRez Studio	
Algorithm for the Analytical Engine	Ada Lovelace	public domain
Ada Lovelace and Book	Moore Allen & Innocent	
MRI Machine	Adobe Stock	education license
JPEG Compression	Adobe	
Noise Cancelling Headphones	Adobe Stock	education license
5G Communication	Adobe Stock	education license
Vibration Measurement	Svantek	
Programmer Icon	Jaime Botero	public domain
Analog Stopwatch	Adobe Stock	education license

Credits

media	source	license
Macbook Pro M2	Apple	
Vax 11/780	Digital Equipment Corp.	
Quantum Circuit	Adobe Stock	education license
Apple M2 Microprocessor	Apple	
Scientific Method	Sue Cahalane	by author
Laboratory Apparatus	pixabay.com	public domain
Dissected Rat	Allen Lew	CC BY 2.0
Tsar Bomba	Rosatom	
The Yoda of Silicon Valley	New York Times	
All Models are Wrong	FreshSpectrum	
Binary Tunnel	Adobe Stock	education license
iPhone 14 Pro Max	Apple	
Babbage's Analytical Engine	Science Museum, London	CC BY-SA 2.0