

<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>

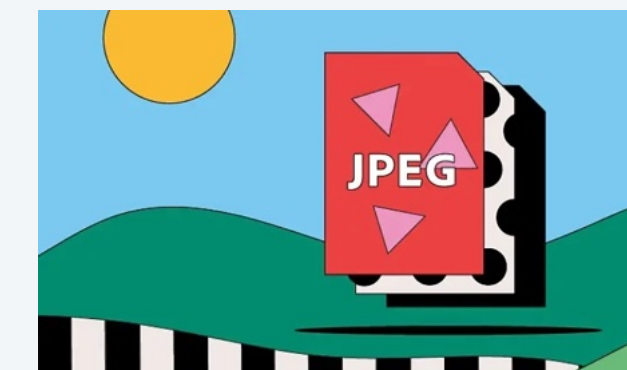
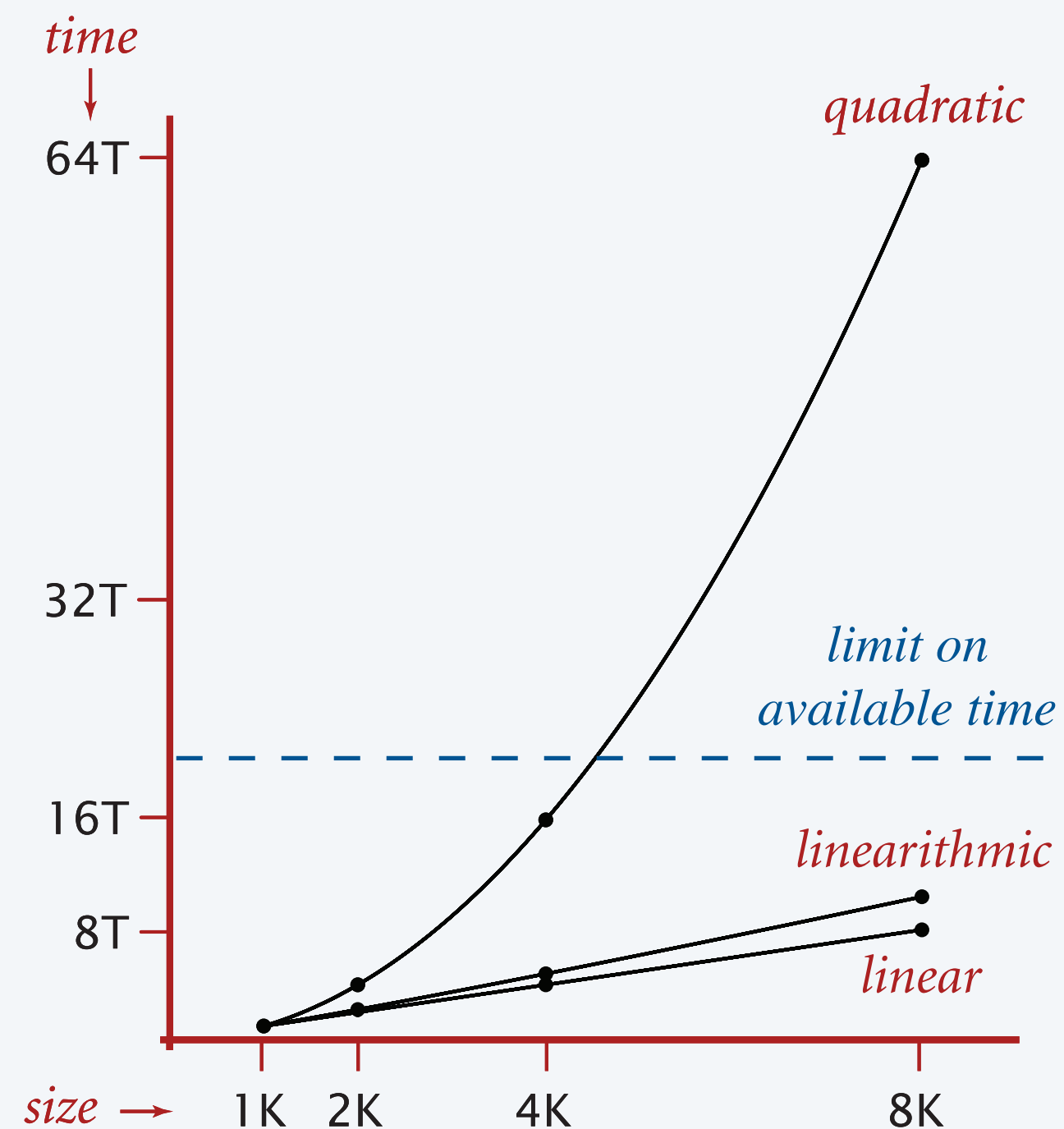
4.1 PERFORMANCE

- ▶ *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**.



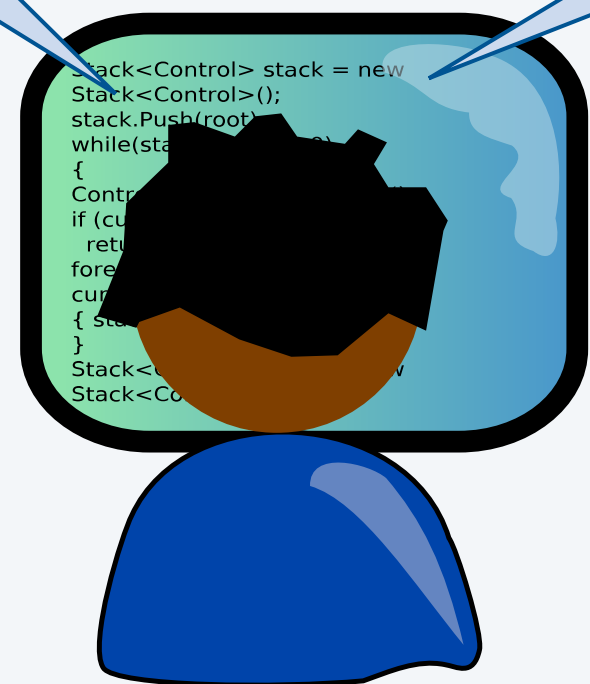
The challenge (modern version)

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?

Why is my program so slow?

Why does it run out of memory?



```
~/cos126/loops> java Factors 11111111111111111111  
2071723 536322235
```

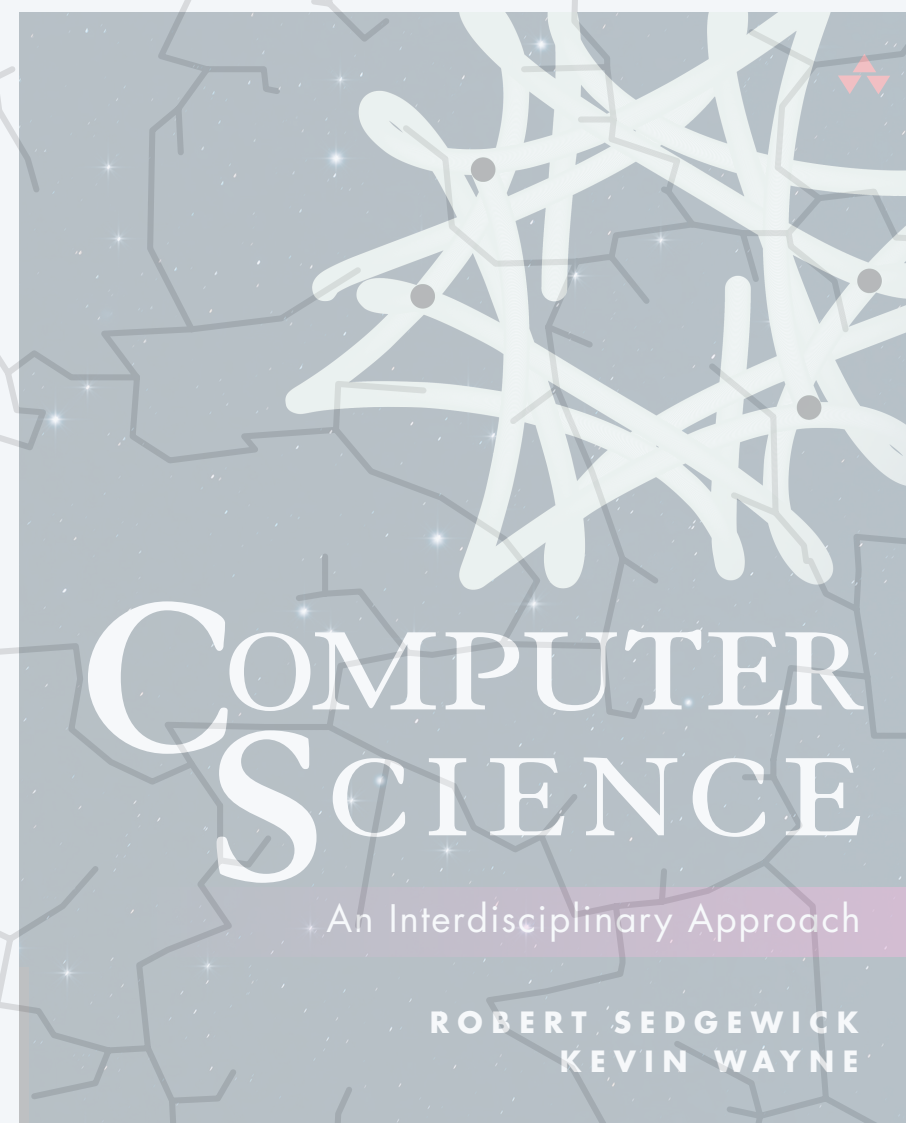
takes a few seconds

```
~/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

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

```
~/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?

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.

```
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)*

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

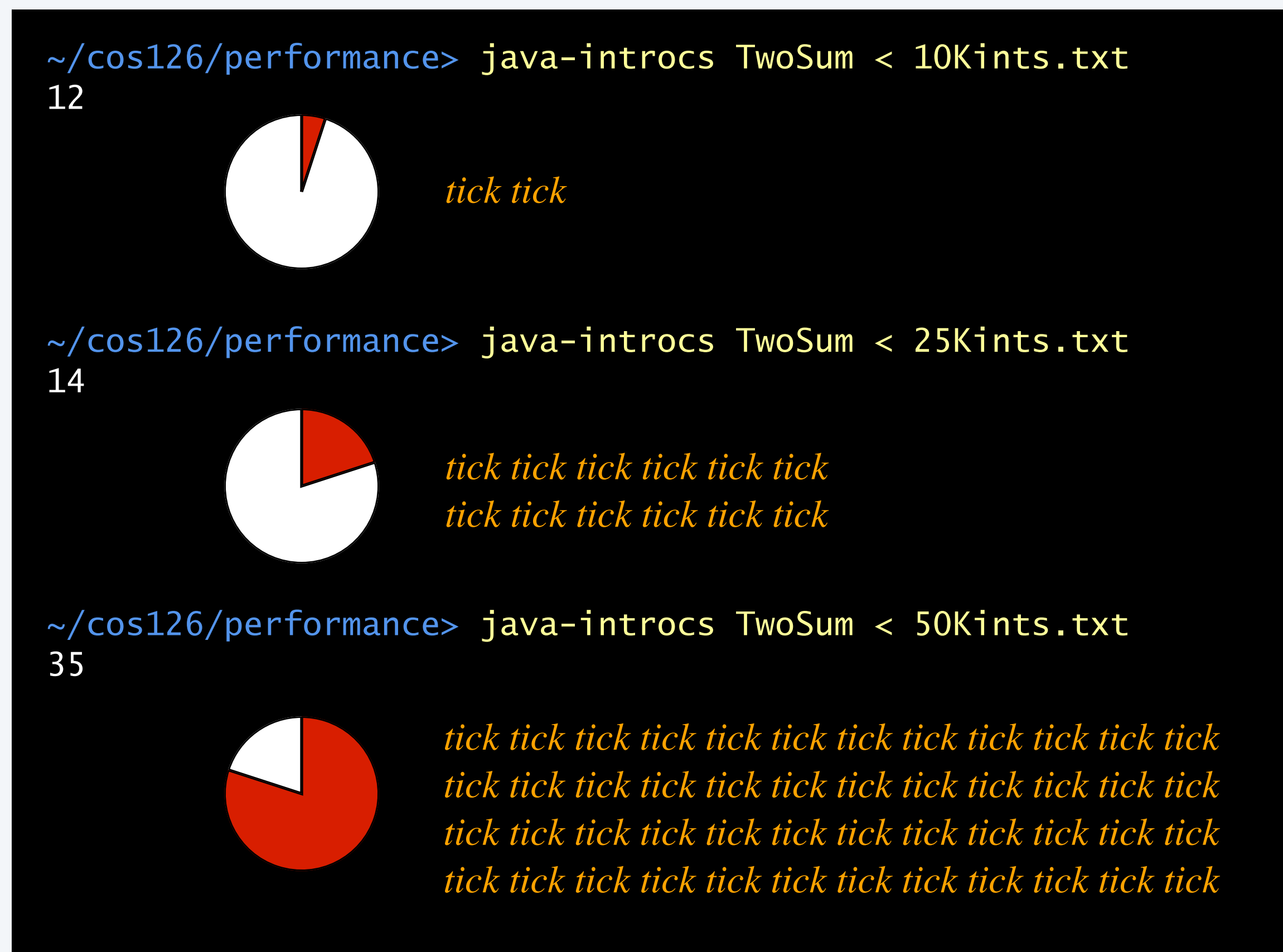
i	j	a[i]	a[j]	sum	
0	1	30	-40	-10	
0	2	30	20	50	
0	3	30	40	70	
0	4	30	-20	10	
1	2	-40	20	-20	
1	3	-40	40	0	✓
1	4	-40	-20	-60	
2	3	20	40	60	
2	4	20	-20	0	✓
3	4	40	-20	20	

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 .



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

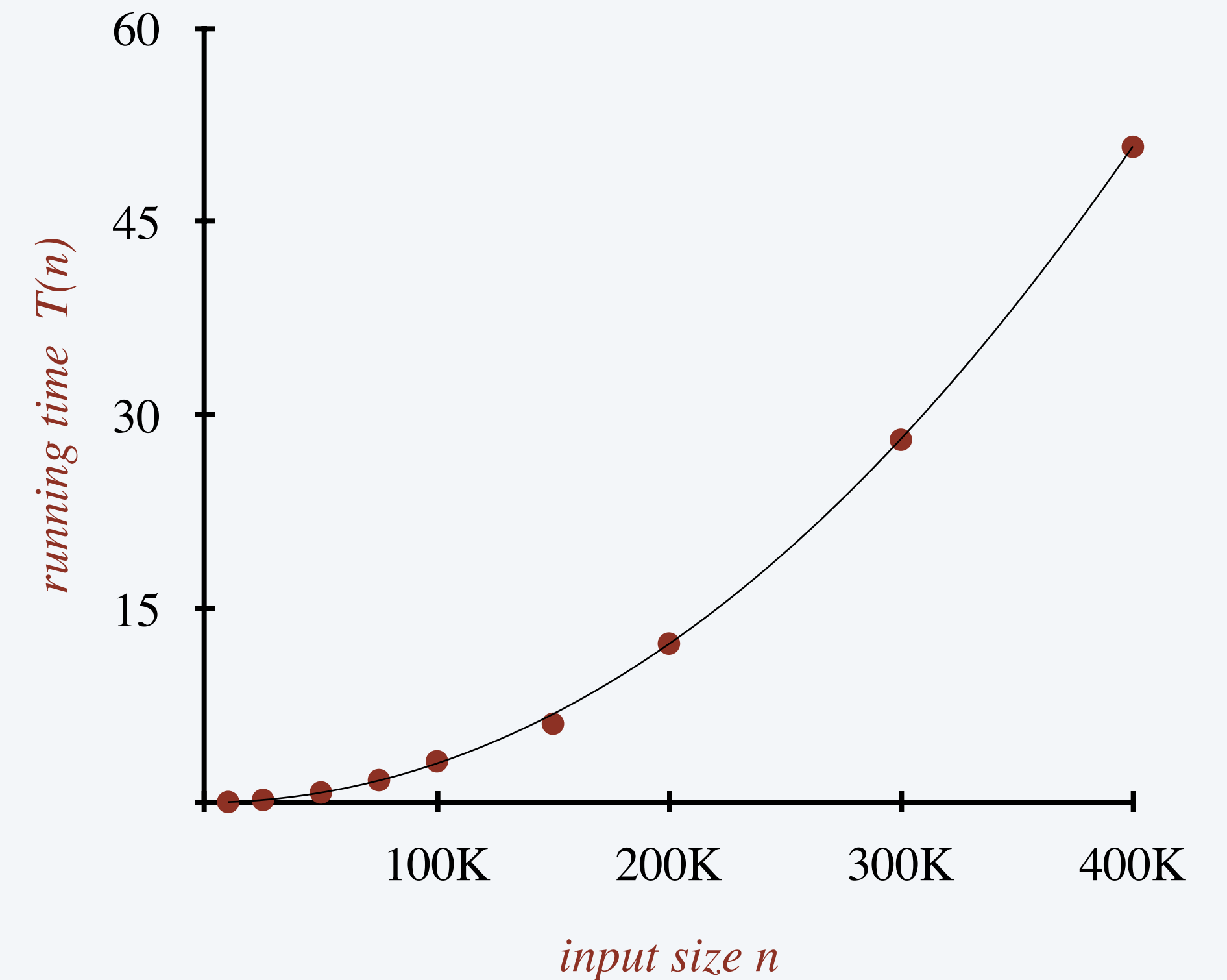


† Apple M2 Pro with 32 GB memory
running OpenJDK 11 on MacOS Ventura

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



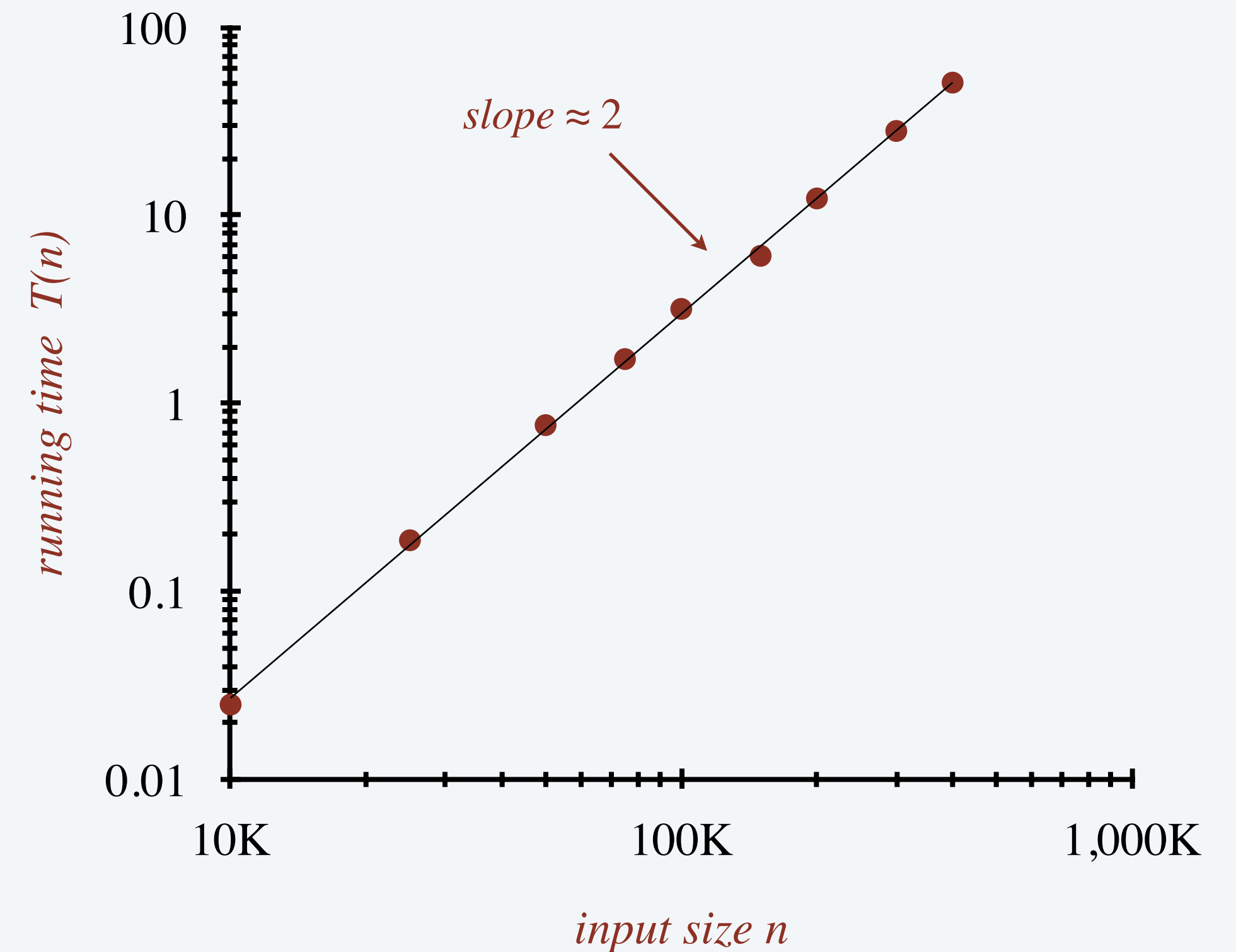
Hypothesis. The running time obeys a **power law**: $T(n) = a \times n^b$ seconds.

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.

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

“quadratic algorithm”
(stay tuned)

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

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

$$\frac{T(n)}{T(n/2)} = \frac{an^b}{a(n/2)^b} = 2^b$$
$$\implies b = \log_2 \frac{T(n)}{T(n/2)}$$

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

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

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

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)*



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)



2020s
(Macbook Pro M2)



futuristic counterexample?
(quantum computer)

Experimental algorithmics

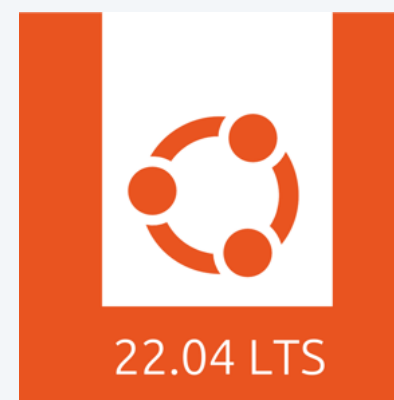
System independent effects.

- Algorithm.
 - Input data.
- ← *determines exponent b
in power law $T(n) = a \times n^b$*

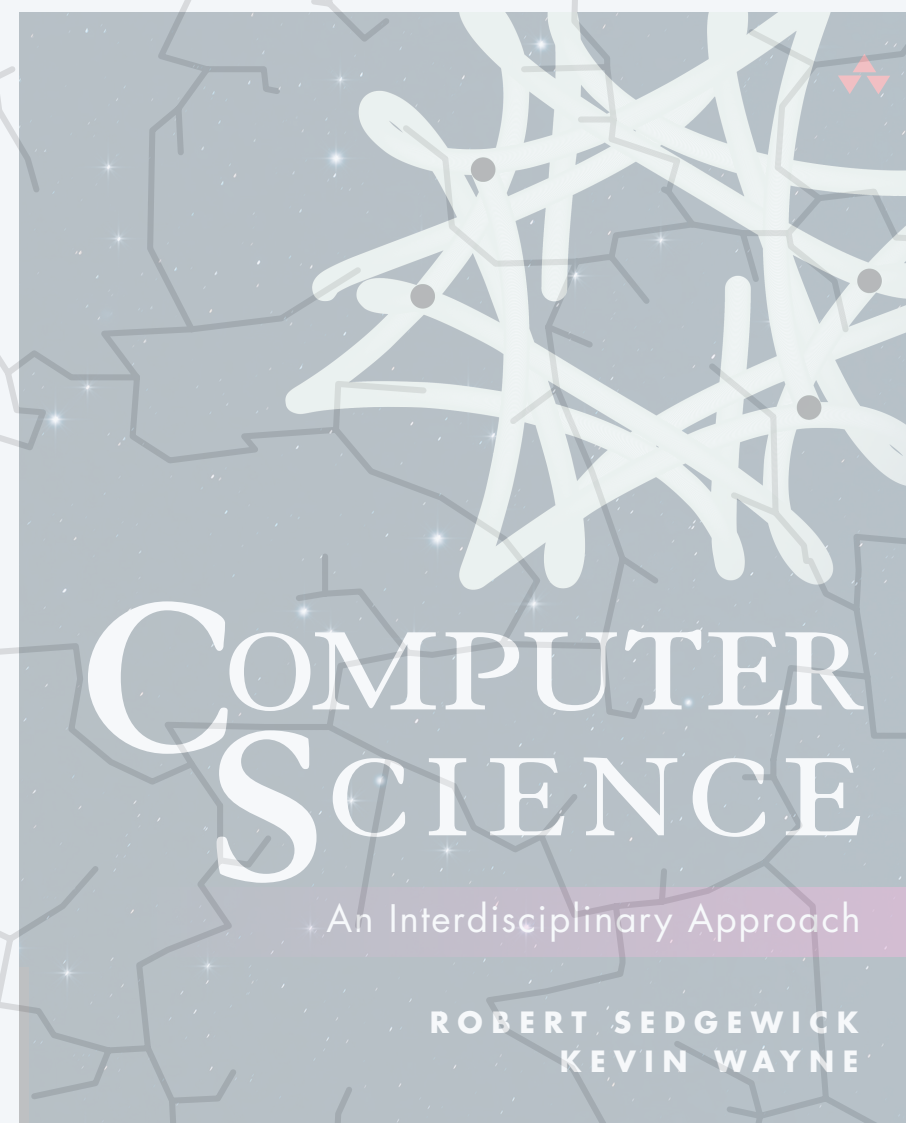
System dependent effects.

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

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



Bad news. Sometimes difficult to get accurate measurements.



<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 \times cost for all operations.

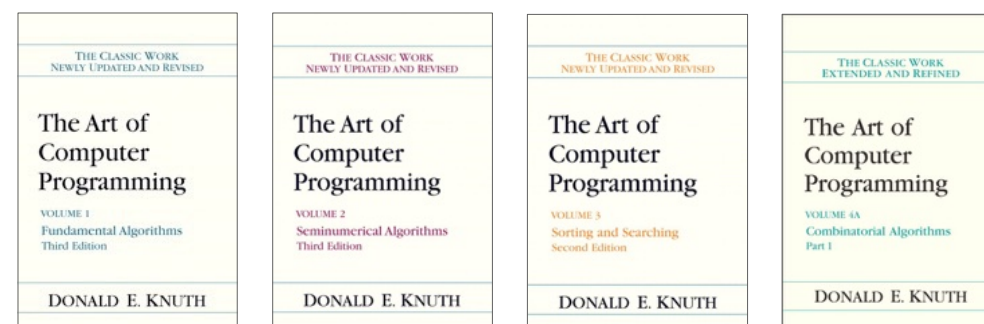
- Frequency depends on algorithm and input data.
- Cost depends on CPU, compiler, operating system, ...

The New York Times

PROFILES IN SCIENCE

The Yoda of Silicon Valley

Donald Knuth, master of algorithms, reflects on 50 years of his opus-in-progress, “The Art of Computer Programming.”



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	
<i>variable declaration</i>	2/5	2	} <i>tedious to count exactly</i>
<i>assignment statement</i>	1/5	2	
<i>less than compare</i>	1/5	$n + 1$	
<i>equal to compare</i>	1/10	n	
<i>array access</i>	1/10	n	
<i>increment</i>	1/10	n to $2n$	

† 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
<i>variable declaration</i>	2/5	2
<i>assignment statement</i>	1/5	2
<i>less than compare</i>	1/5	$n + 1$
<i>equal to compare</i>	1/10	n
<i>array access</i>	1/10	n
<i>increment</i>	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++;
```

$$\begin{array}{c} \text{when } i = 1 \\ \downarrow \\ (n-1) + (n-2) + \dots + 2 + 1 + 0 = \frac{n(n-1)}{2} \\ \uparrow \\ \text{when } i = 0 \end{array} = \binom{n}{2}$$

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)*



Estimate running time as a function of n ?

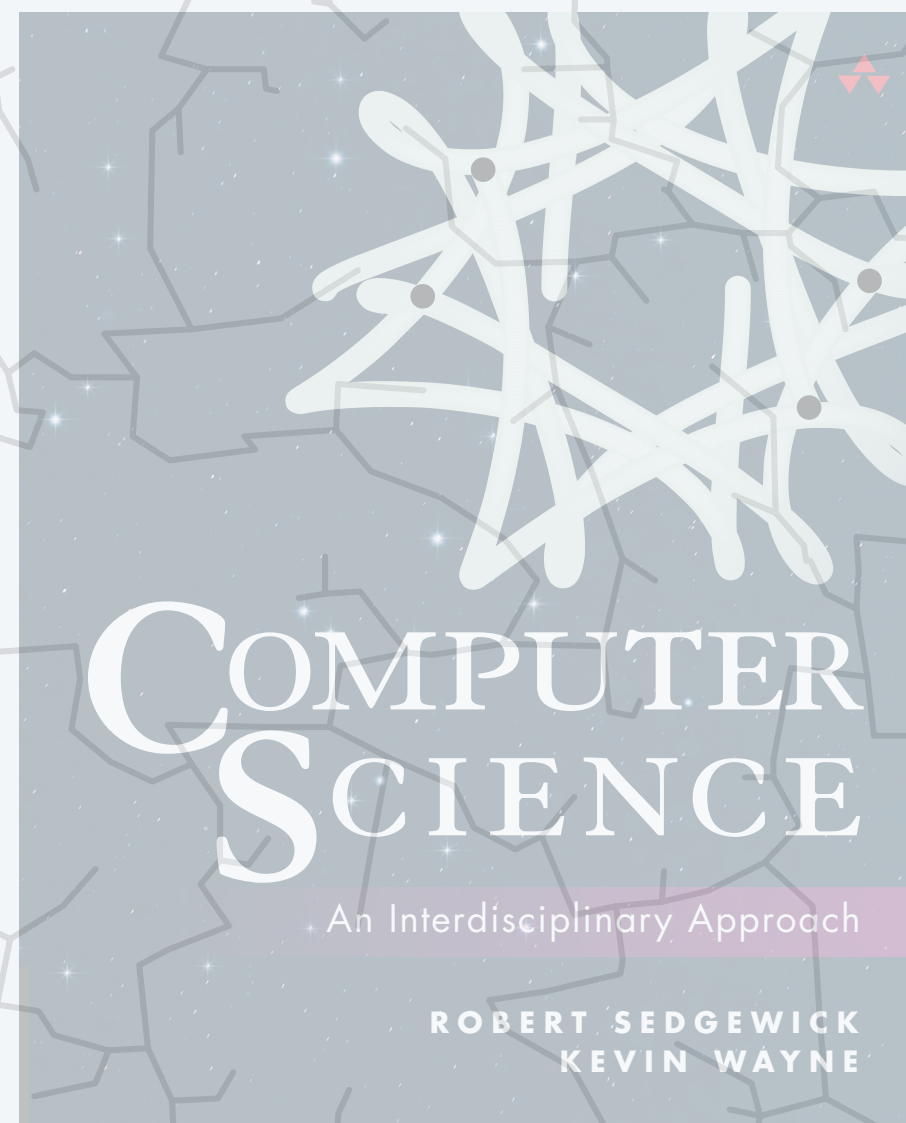
A. $\Theta(n)$

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++;
        }
    }
}
```



<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?

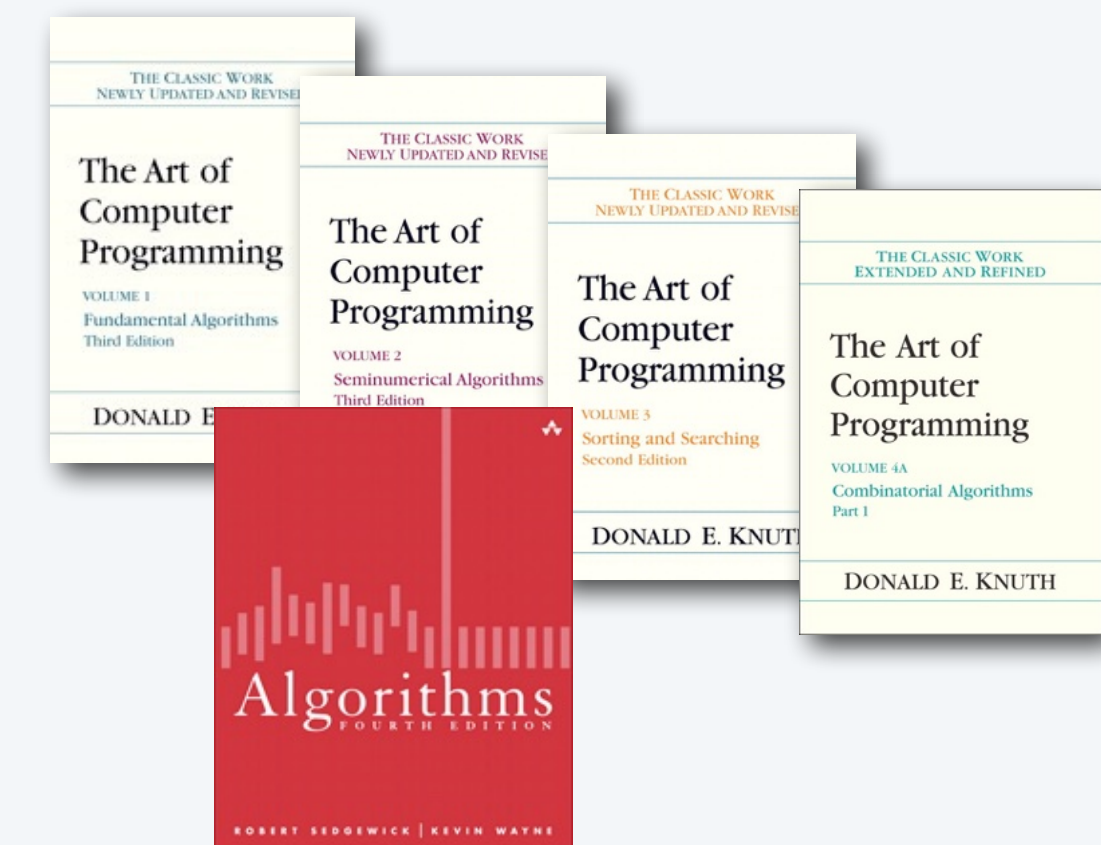
A1. Computer scientists have observed power laws for many many specific algorithms.

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



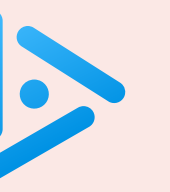
Common order-of-growth classifications

order of growth	emoji	name	typical code framework	description	example	$T(2n) / T(n)$
$\Theta(1)$	💕	constant	<code>a = b + c;</code>	statement	<i>add two numbers</i>	1
$\Theta(\log n)$	😎	logarithmic	<code>for (int i = n; i >= 1; i /= 2) { ... }</code>	divide in half	<i>binary search</i>	~ 1
$\Theta(n)$	😄	linear	<code>for (int i = 0; i < n; i++) { ... }</code>	single loop	<i>find the maximum</i>	2
$\Theta(n \log n)$	😁	linearithmic	<i>mergesort (stay tuned)</i>	divide and conquer	<i>mergesort</i>	~ 2
$\Theta(n^2)$	😞	quadratic	<code>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) { ... }</code>	double loop	<i>check all pairs</i>	4
$\Theta(n^3)$	😡	cubic	<code>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) for (int k = 0; k < n; k++) { ... }</code>	triple loop	<i>check all triples</i>	8
$\Theta(2^n)$	😈	exponential	<i>towers of Hanoi</i>	exhaustive search	<i>check all subsets</i>	2^n

Examples of order-of-growth

computation	implementation	order of growth
<i>dot product</i>	<pre>double sum = 0.0; for (int i = 0; i < n; i++) sum += a[i] * b[i];</pre>	$\Theta(n)$
<i>matrix addition</i>	<pre>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) c[i][j] = a[i][j] + b[i][j];</pre>	$\Theta(n^2)$
<i>matrix multiplication</i>	<pre>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];</pre>	$\Theta(n^3)$
<i>ruler function</i>	<pre>public static int ruler(int n) { if (n == 0) return " "; return ruler(n-1) + n + ruler(n-1); }</pre>	$\Theta(2^n)$

← note: input size
is n^2 , not n

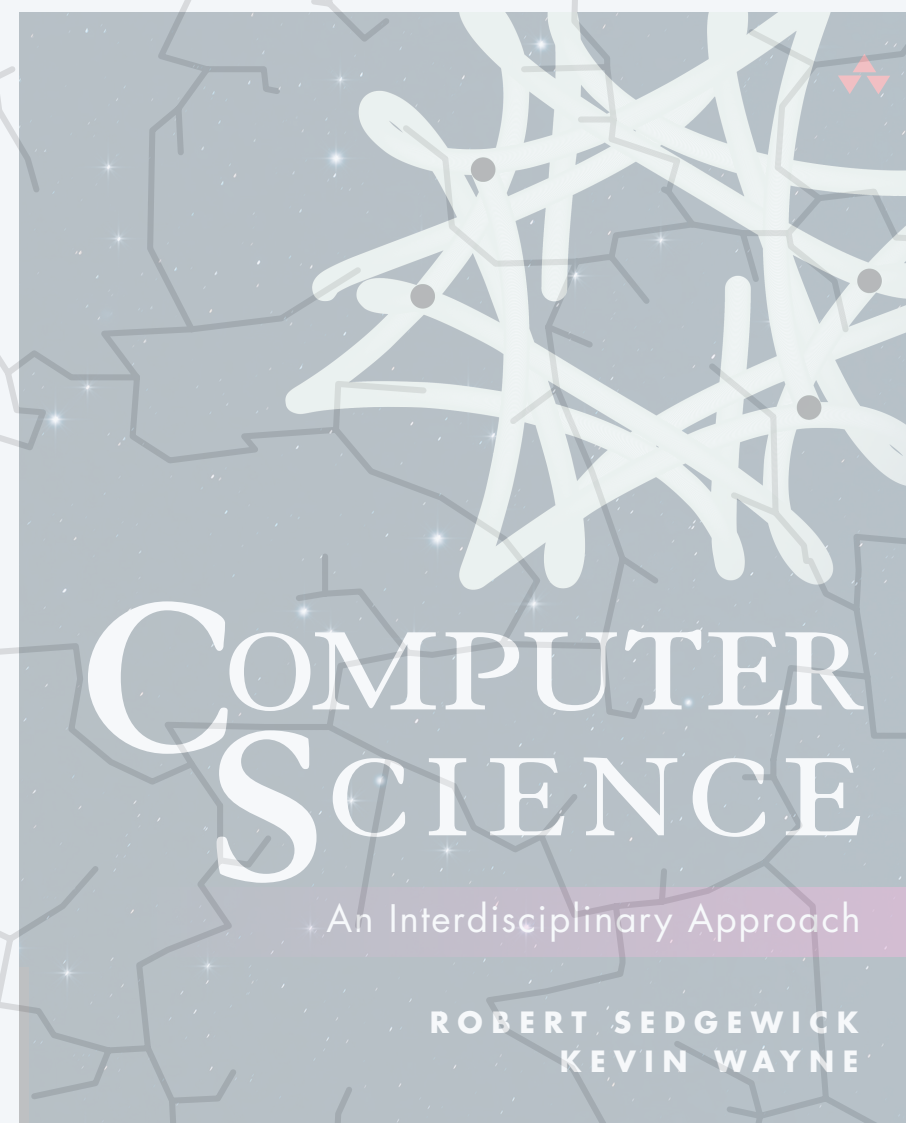


What is order of growth of the running time as a function of n ?

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>



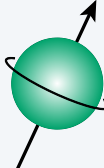



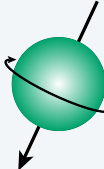

4.1 PERFORMANCE

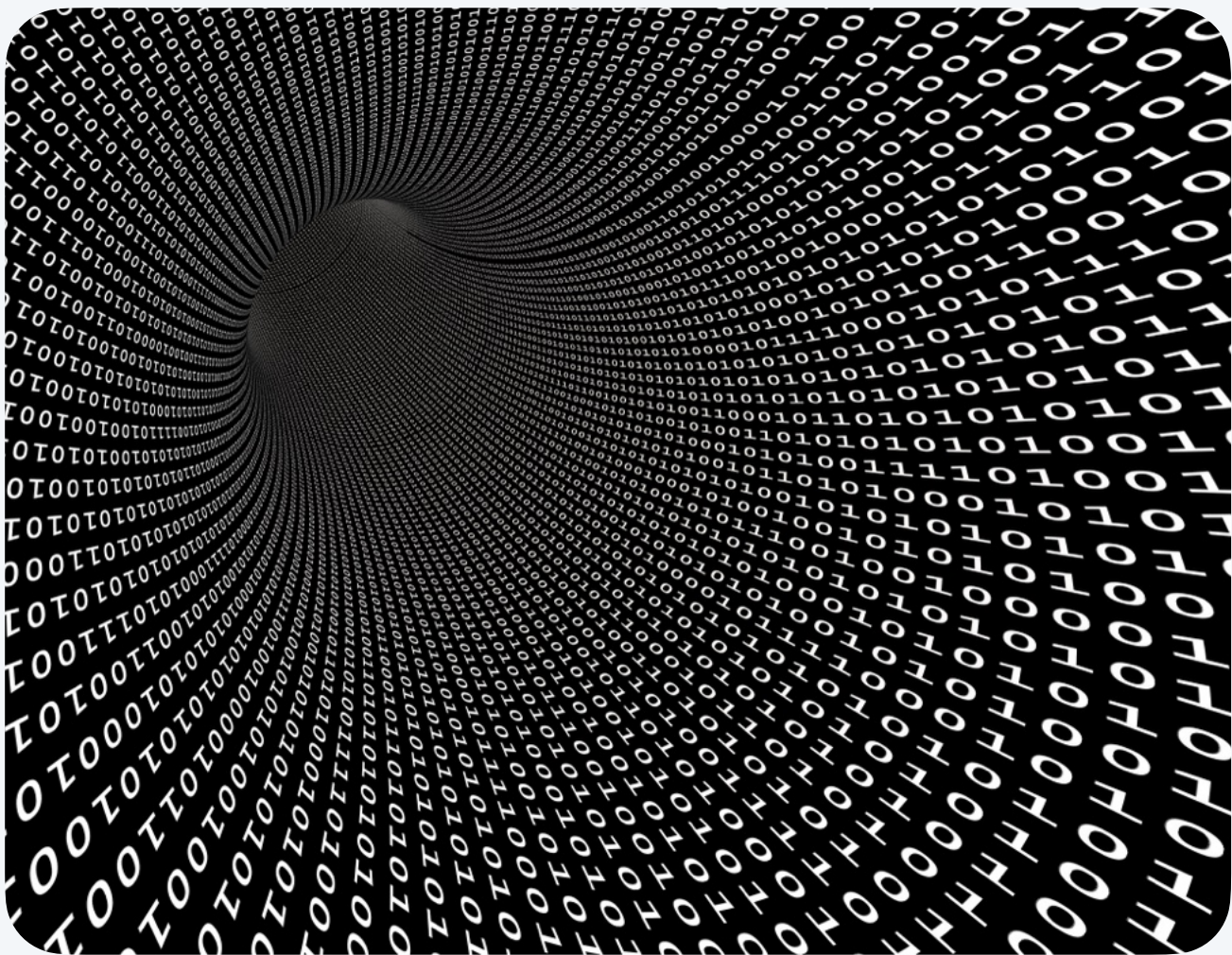
- ▶ *the challenge*
- ▶ *empirical analysis*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *memory usage*

Memory basics

Bit (binary digit). 0 or 1.

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

0				
1				



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

↑
some define using powers of 2
(MB = 2¹⁰ 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)	



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

Credits

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