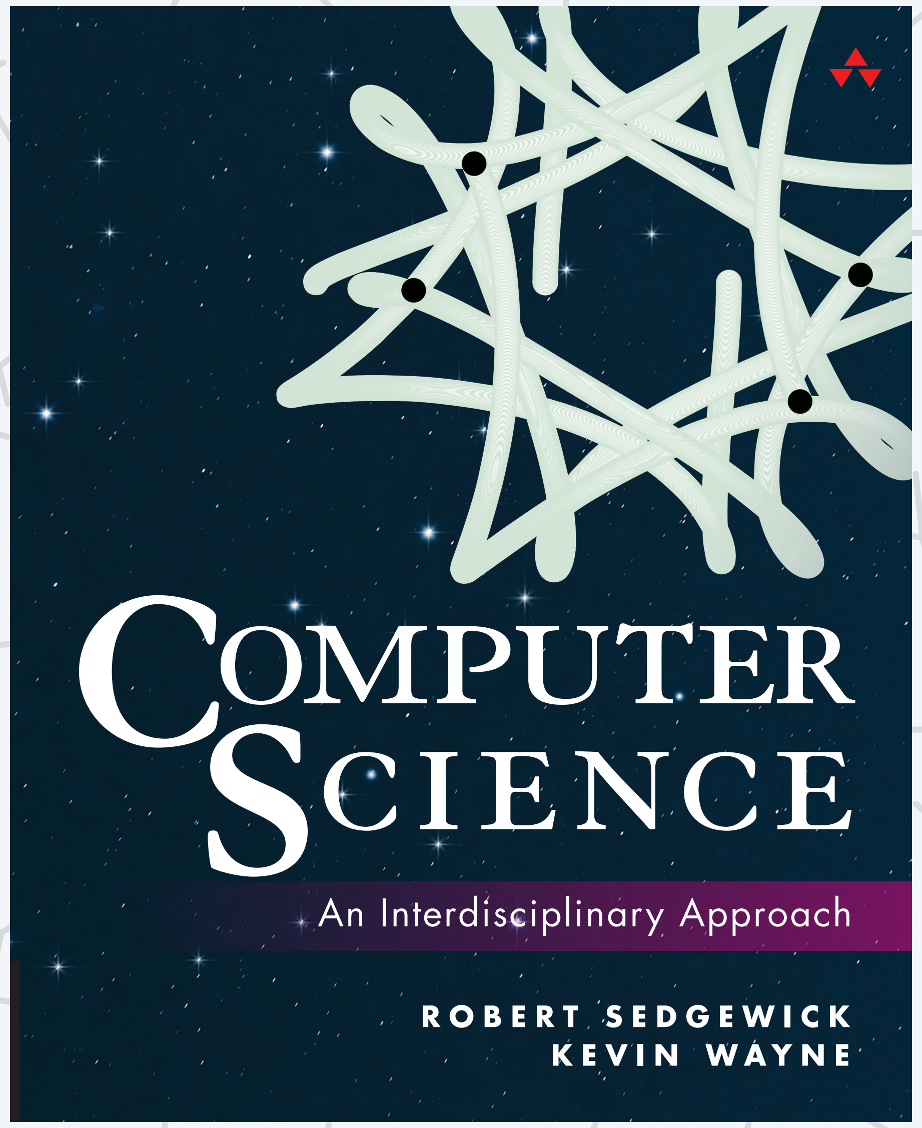
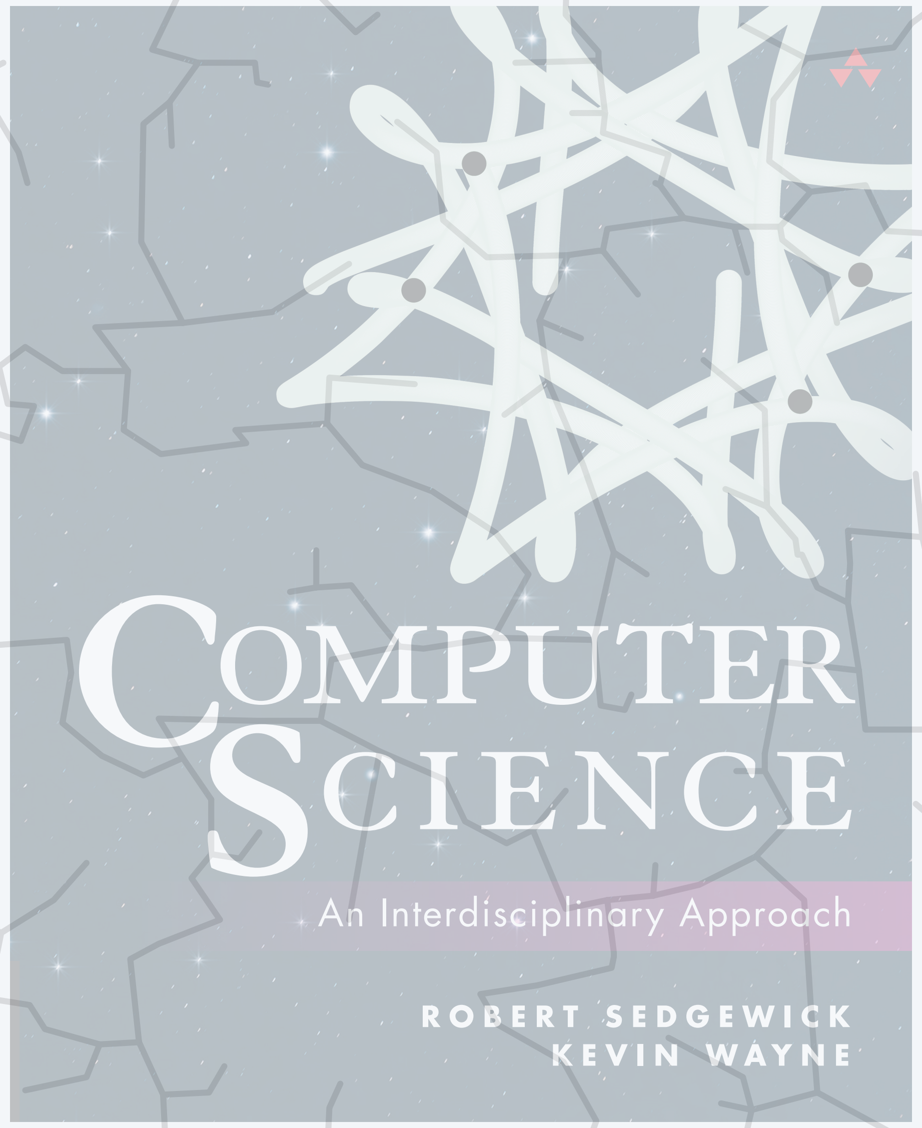


https://introcs.cs.princeton.edu

4.1 Performance

- *intro*
- *empirical analysis*
- *mathematical analysis*
- *notable examples*



https://introcs.cs.princeton.edu

4.1 PERFORMANCE

- ‣ *intro*
- ‣ *empirical analysis*
- ‣ *mathematical analysis*
- ‣ *notable examples*

Performance: quiz 1

Which of the options below best describes "an efficient algorithm" to you?

A. `A.java` processes 1MB (10^6 bytes) in 1 second.

B. `B.java` processes 1GB (10^9 bytes) in 10 seconds.

C. `C.java` processes x bytes in $10{,}000x$ seconds.

D. `D.java` processes x bytes in x^2 seconds.

E. `E.java` processes x bytes in $\dfrac{2^x}{100{,}000}$ seconds.

The runtime function

Suppose `Program.java` can be executed on inputs of arbitrarily large size.

$T(n)$: time (in seconds) taken to run `Program.java` on input of n bytes.

$T(10^6) = 1 \longrightarrow$ **A.** `A.java` processes 1MB (10^6 bytes) in 1 second.

$T(10^9) = 10 \longrightarrow$ **B.** `B.java` processes 1GB (10^9 bytes) in 10 seconds.

$T(n) = 10{,}000 \cdot n \longrightarrow$ **C.** `C.java` processes x bytes in $10{,}000x$ seconds.

$T(n) = n^2 \longrightarrow$ **D.** `D.java` processes x bytes in x^2 seconds.

$T(n) = \dfrac{2^n}{100{,}000} \longrightarrow$ **E.** `E.java` processes x bytes in $\dfrac{2^x}{100{,}000}$ seconds.

What performance *could* mean

Fixed-length input. Input always has length s.

A is better than B if $T_A(s) < T_B(s)$.

Bounded-length input. Input always has length $\leq s$.

A is better than B if $T_A(n) < T_B(n)$ for *all* $n \leq s$.

Unbounded-length input. Input has any length $n > 0$.

A is better than B if $T_A(n) < T_B(n)$ for *all* $n > 0$.

None of these work that well.

Solution. Rate of growth — see next slide!

Many, many more:

- space complexity; ← *P vs. NP*
- polynomial vs. superpolynomial;
- …

Intro: what performance *does* mean (for us)

Rate of growth: leading-order term of $T(n)$, dropping constants.

Examples.

$T(n) = 10{,}000 \cdot n$ →	**C.**	C.java processes x bytes in $10{,}000x$ seconds.	← Rate of growth: n
$T(n) = n^2$ →	**D.**	D.java processes x bytes in x^2 seconds.	← Rate of growth: n^2
$T(n) = \dfrac{2^n}{100{,}000}$ →	**E.**	E.java processes x bytes in $\dfrac{2^x}{100{,}000}$ seconds.	← Rate of growth: 2^n

RoG of $T(n) = n^2 - 100n$ is

RoG of $T(n) = 10n^3 - 600n^2 + 20n - 10{,}000$ is

Intro: what performance *does* mean (for us)

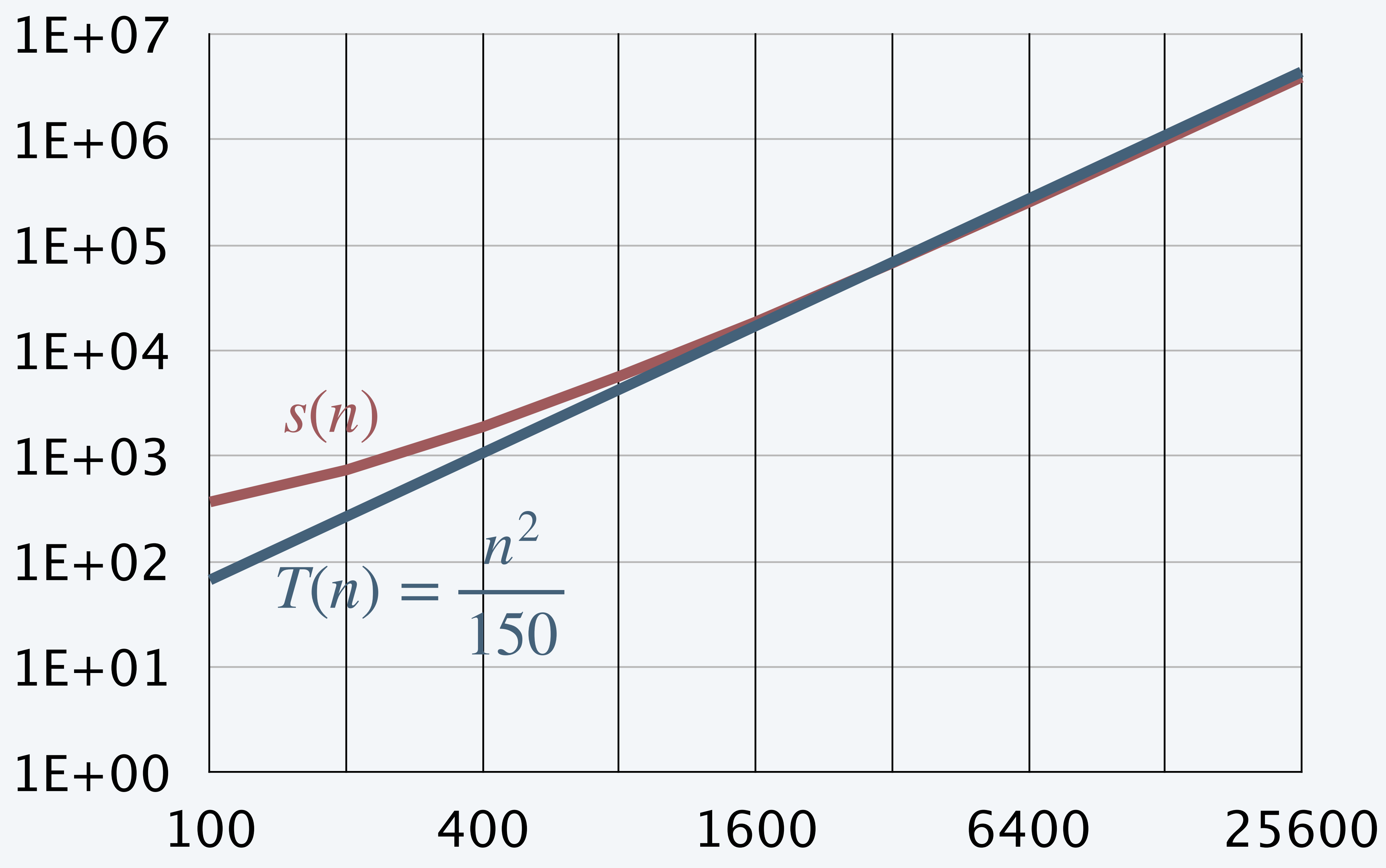
Rate of growth: leading-order term of $T(n)$, dropping constants.

Rate of growth, illustrated. $s(n)$ = size of $n \times n$ PNG image

Image dimensions (pixels)	File size (bytes)
100 x 100	366
200 x 200	736
400 x 400	1,886
800 x 800	5,585
1600 x 1600	18,600
3,200 x 3,200	67,136
6,400 x 6,400	252,917
12,800 x 12,800	984,103
25,600 x 25,600	3,878,458

Dimensions: $\times 2$ between each successive row. File sizes: $\times 2.01$, $\times 2.56$, $\times 2.96$, $\times 3.33$, $\times 3.61$, $\times 3.77$, $\times 3.89$, $\times 3.94$

Loglog plot of side (y axis) vs. dimension (x axis)

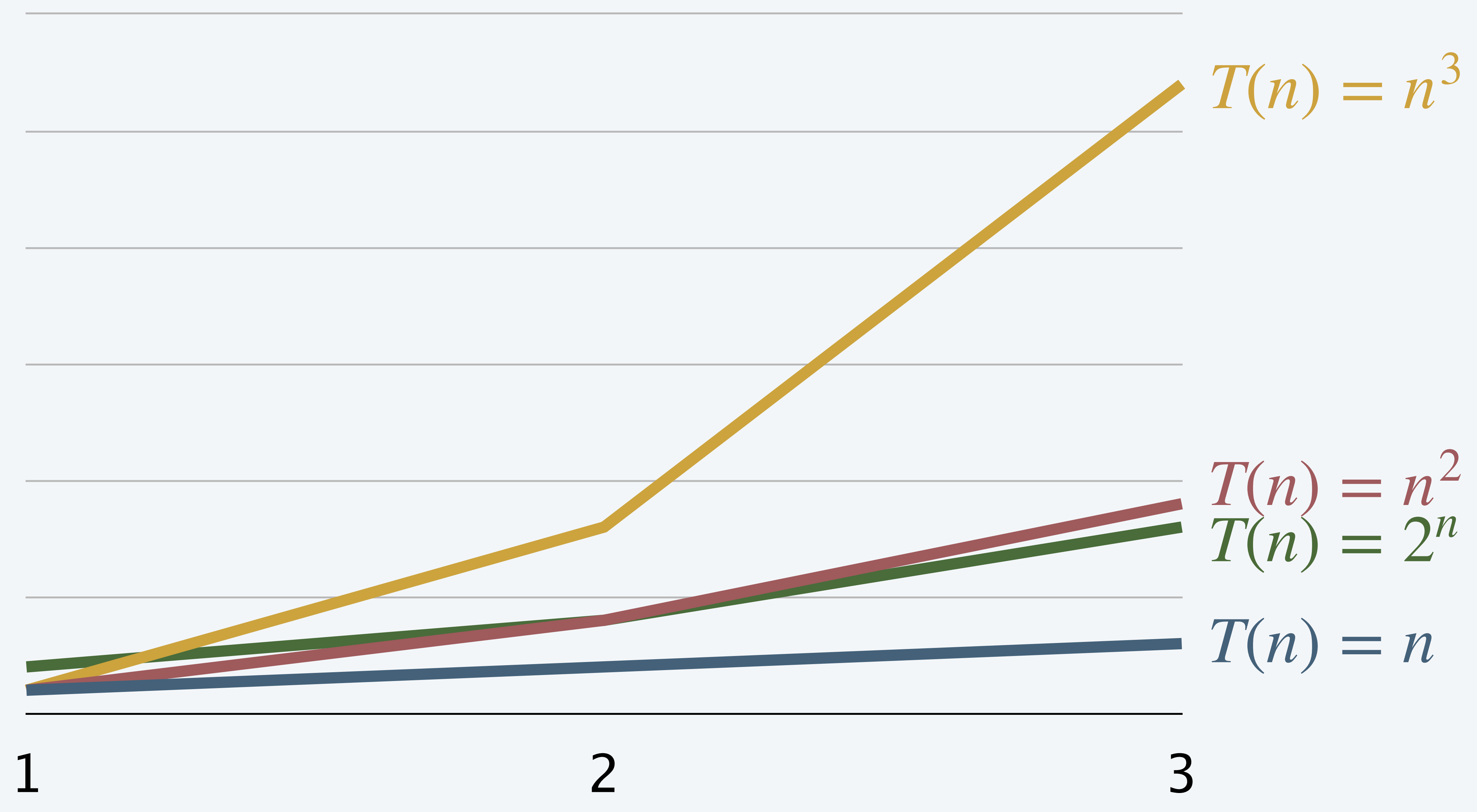


Remark. Table measures *space*, not time. But often connected, as we'll see soon!

Comparing rates of growth

Suppose `Program.java` can be executed on inputs of arbitrarily large size.

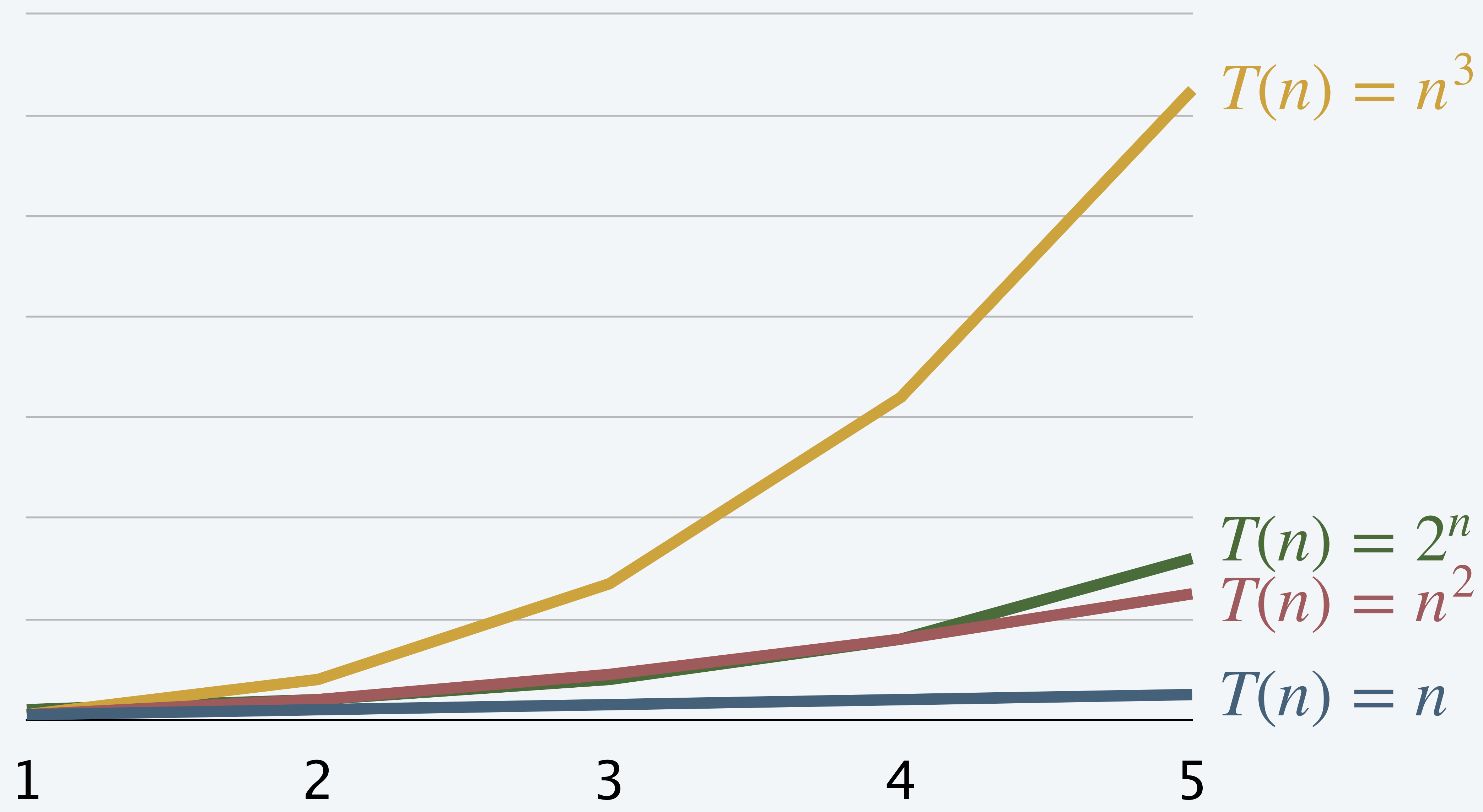
$T(n)$: time taken to run `Program.java` on input of n bytes.



Comparing rates of growth

Suppose `Program.java` can be executed on inputs of arbitrarily large size.

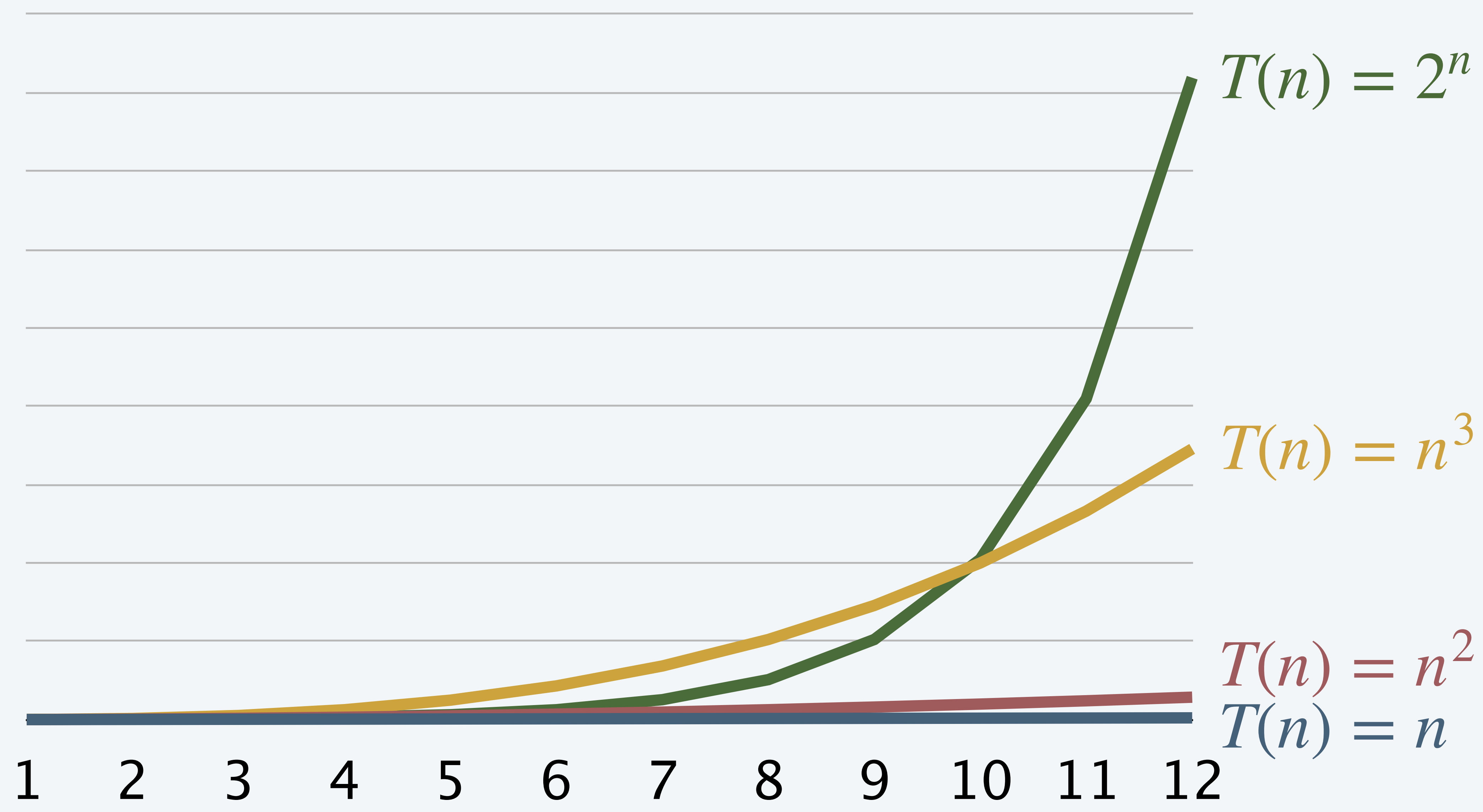
$T(n)$: time taken to run `Program.java` on input of n bytes.



Comparing rates of growth

Suppose `Program.java` can be executed on inputs of arbitrarily large size.

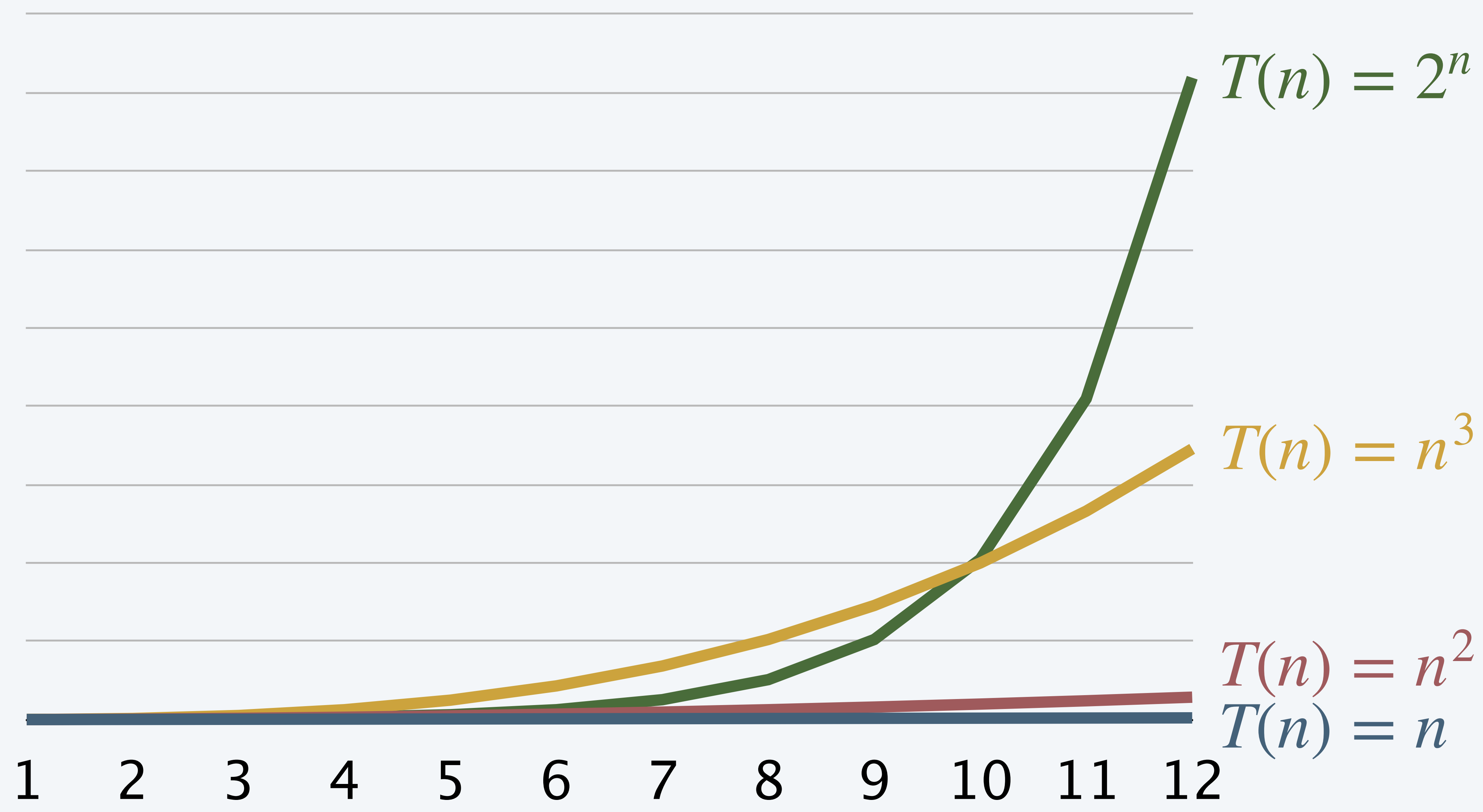
$T(n)$: time taken to run `Program.java` on input of n bytes.



Comparing rates of growth

Suppose `Program.java` can be executed on inputs of arbitrarily large size.

$T(n)$: time taken to run `Program.java` on input of n bytes.



Caveats.

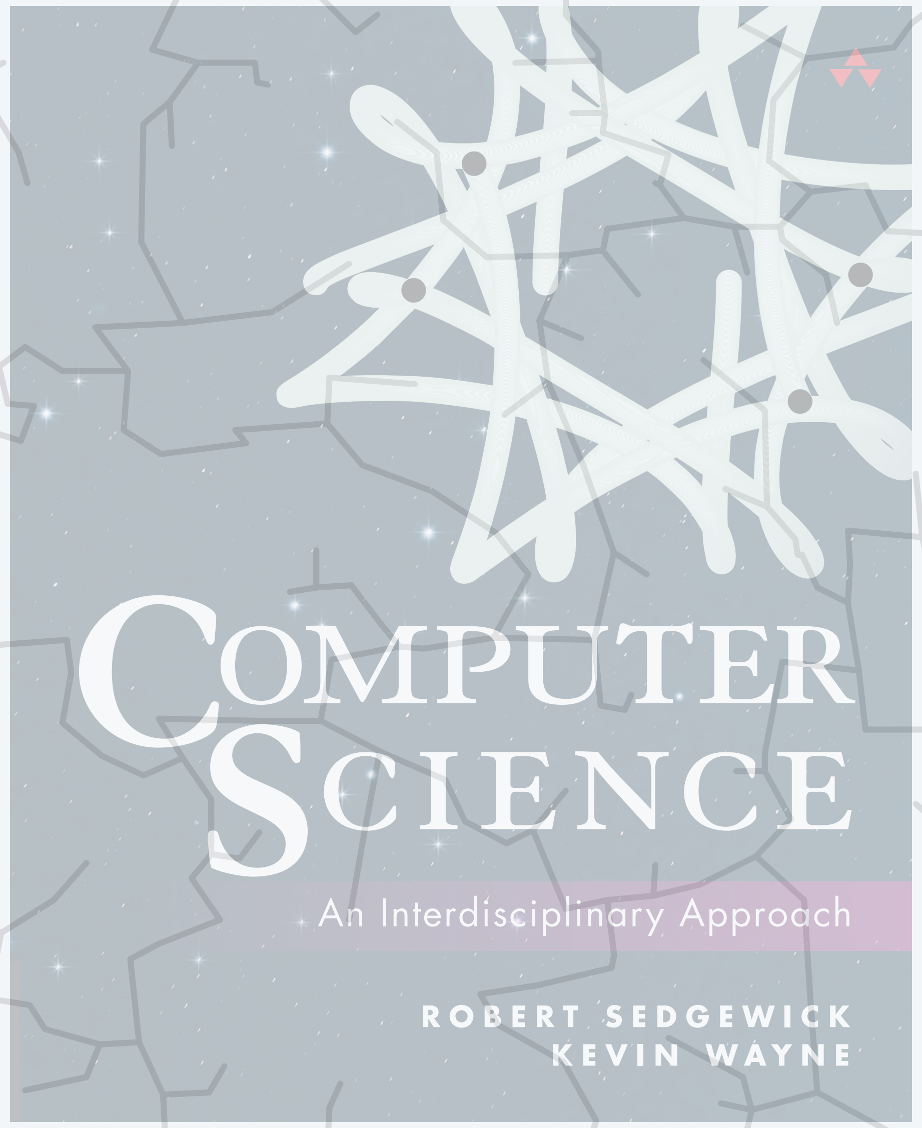
- Input size constrained by hardware & software;
- Runtime varies (a lot) depending on language;
- Time fluctuates across runs on same input;
- …

Solution. Mathematical formalism.

Common orders of growth

formal notation includes Θ, but we'll drop it for simplicity

order of growth	name
$\Theta(1)$	**constant**
$\Theta(\log n)$	**logarithmic**
$\Theta(n)$	**linear**
$\Theta(n \log n)$	**linearithmic**
$\Theta(n^2)$	**quadratic**
$\Theta(n^3)$	**cubic**
$\Theta(n^{\log n})$	**quasipolynomial**
$\Theta(1.1^n)$	**exponential**
$\Theta(2^n)$	**exponential**
$\Theta(n!)$	**factorial**



https://introcs.cs.princeton.edu

4.1 Performance

- intro
- empirical analysis
- mathematical analysis
- notable examples

Checkerboard generator

```
public class Checkerboard {
   public static void main(String[] args) {
      int MIN_LEVEL = 0, MAX_LEVEL = 255;
      int side = Integer.parseInt(args[0]);
      StdPicture.init(side, side);

      for (int col = 0; col < side; col++) {
         boolean black = (col % 2 == 0);
         for (int row = 0; row < side; row++) {
            if (black)
               StdPicture.setRGB(col, row, MIN_LEVEL, MIN_LEVEL, MIN_LEVEL);
            else
               StdPicture.setRGB(col, row, MAX_LEVEL, MAX_LEVEL, MAX_LEVEL);
            black = !black;
         }
      }
      StdPicture.save(side + “x” + side + “.png”);
   }
}
```

picture with width `side` *and height* `side`

first pixel of even/odd cols set to black/white

set to black

set to white

save picture to PNG file

Checkerboard generator

$T(n)$ = time taken to generate an $n \times n$ PNG checkerboard.

Image dimensions (pixels)	Elapsed time (sec)
100 x 100	
200 x 200	
400 x 400	
800 x 800	
1600 x 1600	
3,200 x 3,200	
6,400 x 6,400	
12,800 x 12,800	
25,600 x 25,600	

```
~/> java-introcs Checkerboard 100

~/> java-introcs Checkerboard 200

~/> java-introcs Checkerboard 400

~/> java-introcs Checkerboard 800

~/> java-introcs Checkerboard 1600

~/> java-introcs Checkerboard 3200

~/> java-introcs Checkerboard 6400

~/> java-introcs Checkerboard 12800

~/> java-introcs Checkerboard 25600
```

Remark. Here n is the input itself, not size; difference can be important, but we'll ignore for now.

The doubling method

Assumption. $T(n)$ is a polynomial (can be written as $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ for some k).

1. Choose an initial input.
2. Repeat until it takes too long: ← *e.g., 1 or 2min*
   - Run program on the current input.
   - Record the time elapsed in the run.
   - Double the input.
3. Divide longest by second-longest time, call the result r.
4. Rate of growth is n^k, where 2^k is the power of 2 closest to r.

The math behind it:

$$\frac{T(2n)}{T(n)} = \frac{2^k a_k n^k + \cdots + 2a_1 n + a_0}{a_k n^k + \cdots + a_1 n + a_0}$$

$$= \frac{2^k a_k + \frac{2^{k-1} a_{k-1}}{n} + \cdots + \frac{2a_1}{n^{k-1}} + \frac{a_0}{n^k}}{a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}}$$

$$\xrightarrow{n \to \infty} \frac{2^k a_k}{a_k} = 2^k$$

Variants. Can multiply by another number b instead of 2; then find power of b closest to r.

Nested for loops

Applying the doubling method.

```
long start = System.nanoTime();
for (int i = 0; i < n; i++) {
      // some code
}








long elapsed = (double) (System.nanoTime() - start) / 1_000_000_000;
System.out.println(“Elapsed time: ” + elapsed + “ sec.”);
```

n	**Elapsed time (nanoseconds)**
10^6	
$2 \cdot 10^6$	
$4 \cdot 10^6$	
$8 \cdot 10^6$	
$16 \cdot 10^6$	
$32 \cdot 10^6$	

Nested for loops

Applying the doubling method.

```
long start = System.nanoTime();
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      // some code
   }
}




long elapsed = (double) (System.nanoTime() - start) / 1_000_000_000;
System.out.println("Elapsed time: " + elapsed + " sec.");
```

n	Elapsed time (nanoseconds)
2,000	
4,000	
8,000	
16,000	
32,000	
64,000	

Nested for loops

Applying the doubling method.

```
long start = System.nanoTime();
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
      // some code
      }
   }
}


long elapsed = (double) (System.nanoTime() - start) / 1_000_000_000;
System.out.println("Elapsed time: " + elapsed + " sec.");
```

n	Elapsed time (nanoseconds)
50	
100	
200	
400	
800	
1,600	

Nested for loops

Applying the doubling method.

```
long start = System.nanoTime();
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
         for (int l = 0; l < n; l++) {
            // some code
         }
      }
   }
}
long elapsed = (double) (System.nanoTime() - start) / 1_000_000_000;
System.out.println("Elapsed time: " + elapsed + " sec.");
```

n	Elapsed time (nanoseconds)
10	
20	
40	
80	
160	
320	

Performance: quiz 2

As n grows, what does `ratio` converge to?

A. 2

B. 3

C. 4

D. 9

E. 16

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      // some code
   }
}
```

Performance: quiz 3

As n grows, what does `ratio` converge to?

A. 2

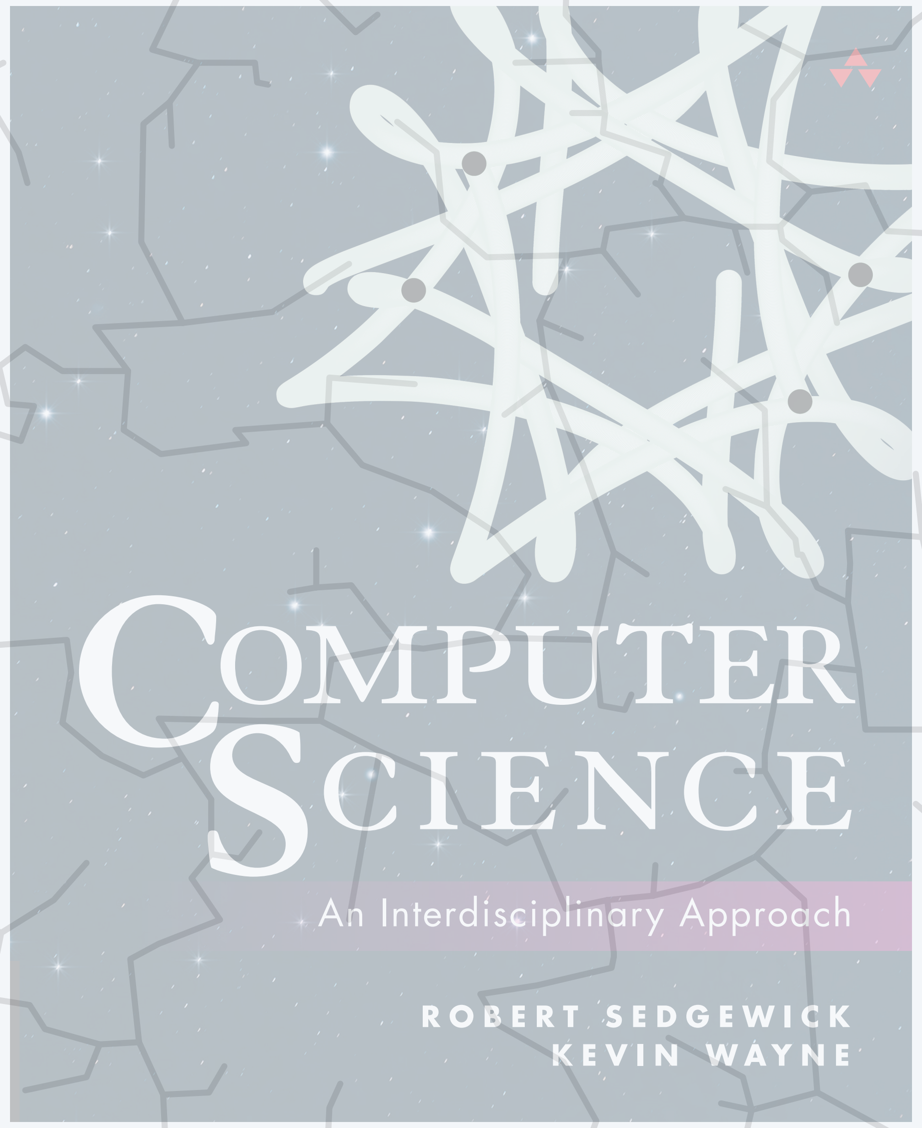
B. 3

C. 4

D. 9

E. 16

```
for (int i = 0; i < n; i++) {
   // some code
}
for (int j = 0; j < n; j++) {
   // some code
}
```



https://introcs.cs.princeton.edu

4.1 PERFORMANCE

- intro
- empirical analysis
- mathematical analysis
- notable examples

Mathematical analysis

Elementary operations:

not elementary: `StdPicture.read()`, `StdAudio.play()`, etc.

- Declaring/assigning variable;
- Printing fixed-length string;
- Arithmetic operation;
- …

Count # of elementary operations. Program tracing!

```
for (int i = 0; i < n; i++) {
   // some elementary operations
}
```

i	# of iterations
0	1
1	2
2	3
3	4
⋮	⋮
$n - 1$	n

Mathematical analysis

Elementary operations:

- Declaring/assigning variable;
- Printing fixed-length string;
- Arithmetic operation;
- …

Count # of elementary operations. Program tracing!

```
for (int i = 1; i <= n; i *= 2) {
   // some elementary operations
}
```

i	# of iterations
1	1
2	2
4	3
8	4
$\vdots$	$\vdots$
n	$1 + \log_2 n$

Mathematical analysis

Elementary operations:

- Declaring/assigning variable;
- Printing fixed-length string;
- Arithmetic operation;
- …

Count # of elementary operations. Program tracing!

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      // some elementary operations
   }
}
```

	i	*j*	*# of iterations*
	0	0	1
	0	1	2
n iterations	0	2	3
	0	3	4
	⋮	⋮	⋮
	0	$n - 1$	n
	1	0	$n + 1$
n iterations	1	1	$n + 2$
	⋮	⋮	⋮
	1	$n - 1$	$2n$
	2	0	$2n + 1$
n iterations	⋮	⋮	⋮
	2	$n - 1$	$3n$
	3	0	$3n + 1$
n iterations	⋮	⋮	⋮
	3	$n - 1$	$4n$
	⋮	⋮	⋮
	$n - 1$	0	$(n-1)n + 1$
n iterations	⋮	⋮	⋮
	$n - 1$	$n - 1$	n^2

Mathematical analysis

Elementary operations:

- Declaring/assigning variable;
- Printing fixed-length string;
- Arithmetic operation;
- …

Count # of elementary operations. Program tracing!

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      // some elementary operations
   }
}
```

n^2 iterations

i	j	# of iterations
0	0	1
0	1	2
0	2	3
0	3	4
⋮	⋮	⋮
0	$n - 1$	n
1	0	$n + 1$
1	1	$n + 2$
⋮	⋮	⋮
1	$n - 1$	$2n$
2	0	$2n + 1$
⋮	⋮	⋮
2	$n - 1$	$3n$
3	0	$3n + 1$
⋮	⋮	⋮
3	$n - 1$	$4n$
⋮	⋮	⋮
$n - 1$	0	$(n-1)n + 1$
⋮	⋮	⋮
$n - 1$	$n - 1$	n^2

Mathematical analysis

Elementary operations:

- Declaring/assigning variable;
- Printing fixed-length string;
- Arithmetic operation;
- …

Count # of elementary operations. Program tracing!

```
for (int i = 0; i < n; i++) {
   for (int j = i; j < n; j++) {
      // some elementary operations
   }
}
```

initialized to i

	i	j	# of iterations
	0	0	1
	0	1	2
n iterations	0	2	3
	0	3	4
	⋮	⋮	⋮
	0	n - 1	n
	1	1	$n + 1$
$n - 1$ iterations	1	2	$n + 2$
	⋮	⋮	⋮
	1	n - 1	$2n$ - 1
	2	2	2n
$n - 2$ iterations	⋮	⋮	⋮
	2	n - 1	$3n$ - 3
	3	3	$3n$ - 2
$n - 3$ iterations	⋮	⋮	⋮
	3	n - 1	$4n$ - 6
	⋮	⋮	⋮
1 iteration	n - 1	n - 1	$(n^2 + n)/2$

Mathematical analysis

The math behind it:

Call $N = n + (n-1) + \cdots + 2 + 1.$

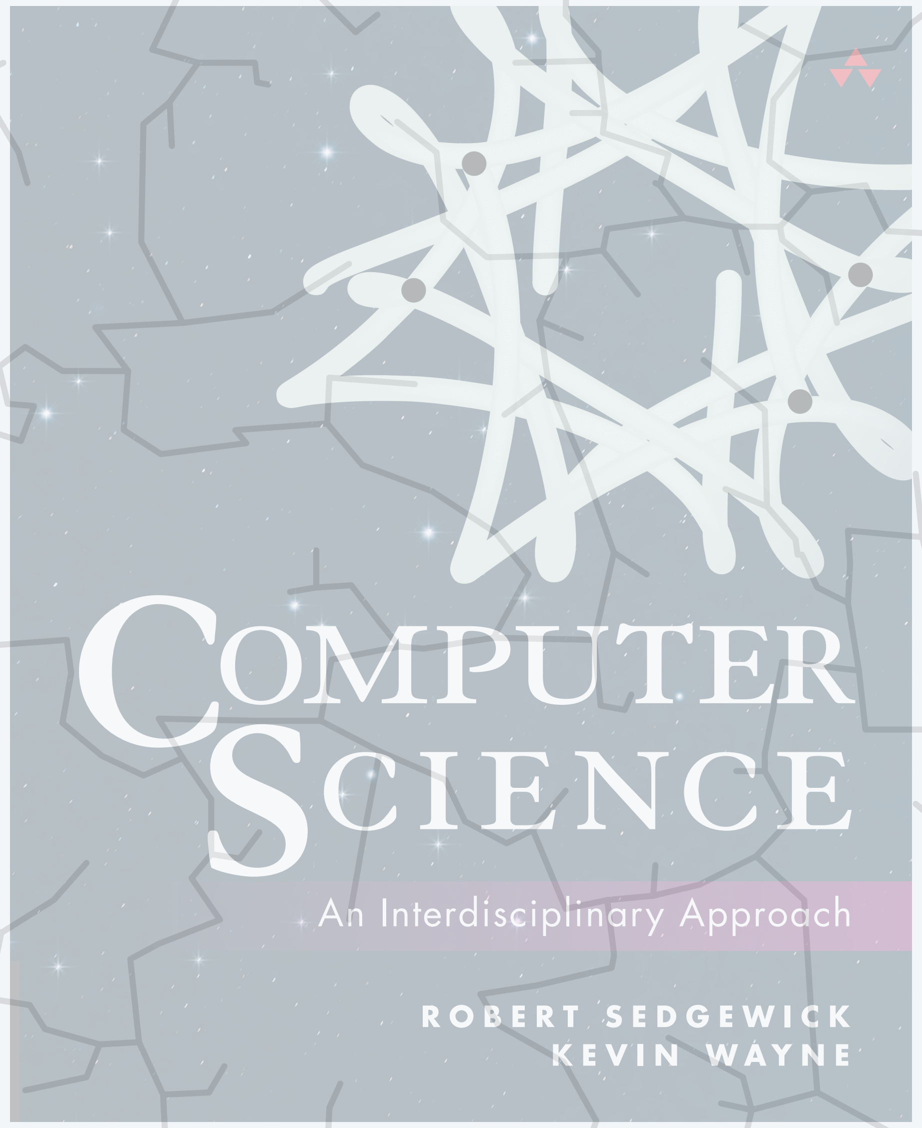
Then $2N = \begin{array}{cccccccccc} & n & + & n-1 & + & \cdots & + & 2 & + & 1 \\ + & 1 & + & 2 & + & \cdots & + & n-1 & + & n \end{array} = n \cdot (n+1).$

Therefore, $N = \dfrac{n \cdot (n+1)}{2}.$

```
for (int i = 0; i < n; i++) {
   for (int j = i; j < n; j++) {
      // some elementary operations
   }
}
```

$\dfrac{n^2}{2} + \dfrac{n}{2}$ iterations

i	j	# of iterations
0	0	1
0	1	2
0	2	3
0	3	4
⋮	⋮	⋮
0	n - 1	n
1	1	n + 1
1	2	n + 2
⋮	⋮	⋮
1	n - 1	2n - 1
2	2	2n
⋮	⋮	⋮
2	n - 1	3n - 3
3	3	3n - 2
⋮	⋮	⋮
3	n - 1	4n - 6
⋮	⋮	⋮
n - 1	n - 1	$(n^2 + n)/2$



https://introcs.cs.princeton.edu

4.1 Performance

- intro
- empirical analysis
- mathematical analysis
- notable examples

Integer factorization

Goal. Given a positive integer n, find its prime factorization.

$$98 = 2 \times 7 \times 7 \qquad 3{,}757{,}208 = 2 \times 2 \times 2 \times 7 \times 13 \times 13 \times 397 \qquad 11{,}111{,}111{,}111{,}111{,}111 = 2{,}071{,}723 \times 5{,}363{,}222{,}357$$

Grade-school factoring algorithm.

FACTOR(*n*)

Consider each potential divisor d between 2 and n:

- ***while*** d is a divisor of n:
  - ***print*** d
  - $n \leftarrow n \,/\, d$

Critical application. Cryptography.  ← *security of internet commerce relies on difficulty of factoring very large integers*

Integer factorization

```
public class Factors {
   public static void main(String[] args) {
      long n = Long.parseLong(args[0]);
      for (long d = 2; d <= n; d++) {
         while (n % d == 0) {
            System.out.print(d + " ");
            n = n / d;
         }
      }
      System.out.println();
   }
}
```

try all possible divisors d

if d is a divisor, factor it out

```
~/cos126/loops> java Factors 98
2 7 7

~/cos126/loops> java Factors 3757208
2 2 2 7 13 13 397

~/cos126/loops> java Factors 97
97

~/cos126/loops> java Factors 11111111111111111
2071723 5363222357
```

takes a few seconds

Remark. Way too slow to break cryptography. (Input *size* is # of digits, so exponential runtime!)

can be sped up substantially by stopping when $d > \sqrt{n}$ (but still way too slow)

How difficult can it be?

Imagine a galactic computer…

- With as many processors as electrons in the universe.
- Each processor having the power of today's supercomputers.
- Each processor working for the lifetime of the universe.

quantity	estimate
electrons in universe	10^{79}
instructions per second	10^{18}
age of universe in seconds	10^{17}

Q. Could galactic computer run `Factors.java` on a $1{,}000$-digit (prime) number?

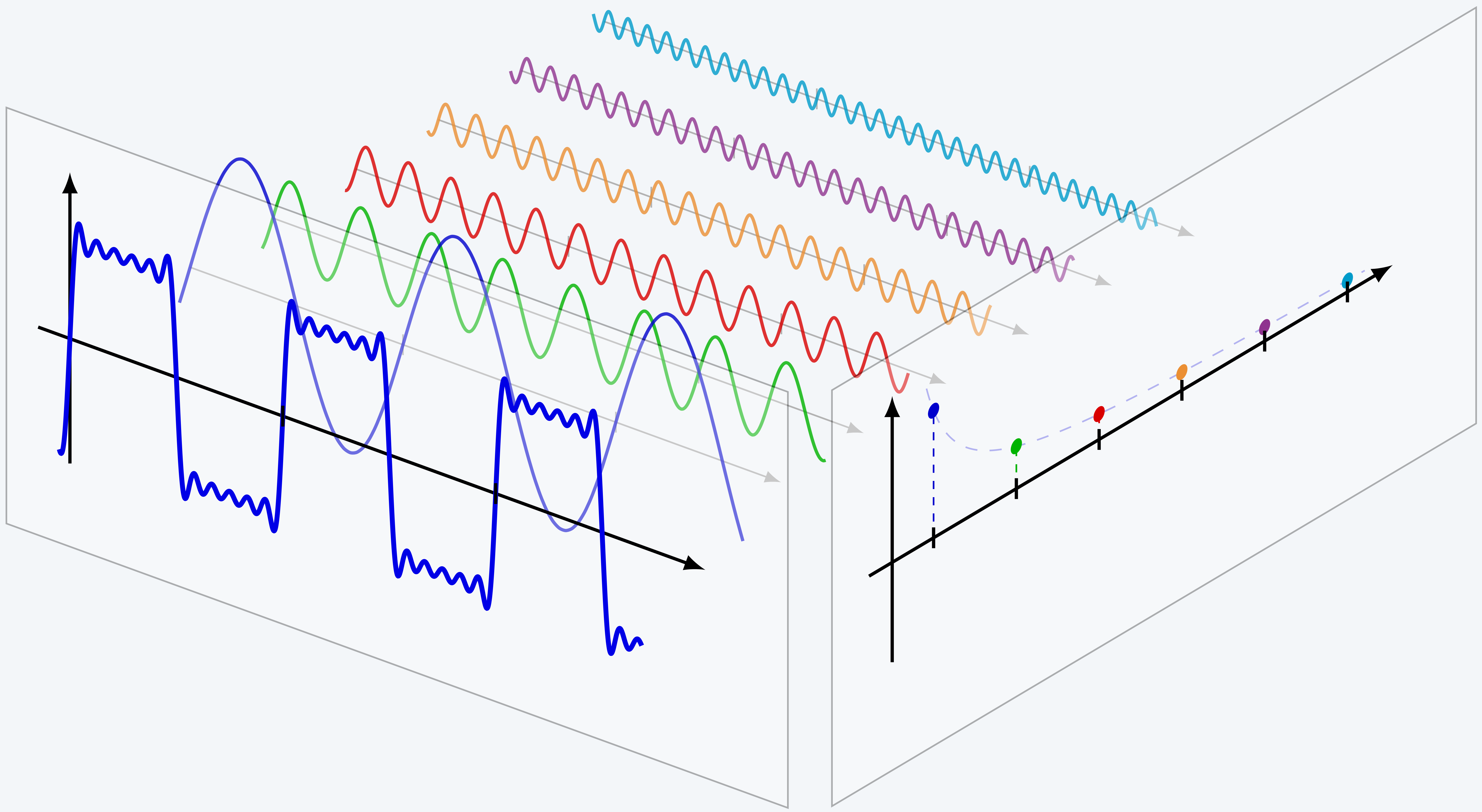
A. Not even close: $10^{1000} \gg 10^{79} \cdot 10^{18} \cdot 10^{17} = 10^{114}$.

Lesson. Exponential growth dwarfs technological change.

Fast Fourier Transform

Critical application. Signal processing. ← *including Wi-Fi, 5G, JPEG, MP3…*

> *"the most important numerical algorithm of our lifetime"* — *Gilbert Strang*



Fast Fourier Transform

Critical application. Signal processing.

In computational math: Multiplying n-digit numbers.

- Grade-school algorithm: n^2 time.
- Schönhage–Strassen (SS) algorithm: $n \cdot \log n \cdot \log\log n$ time!

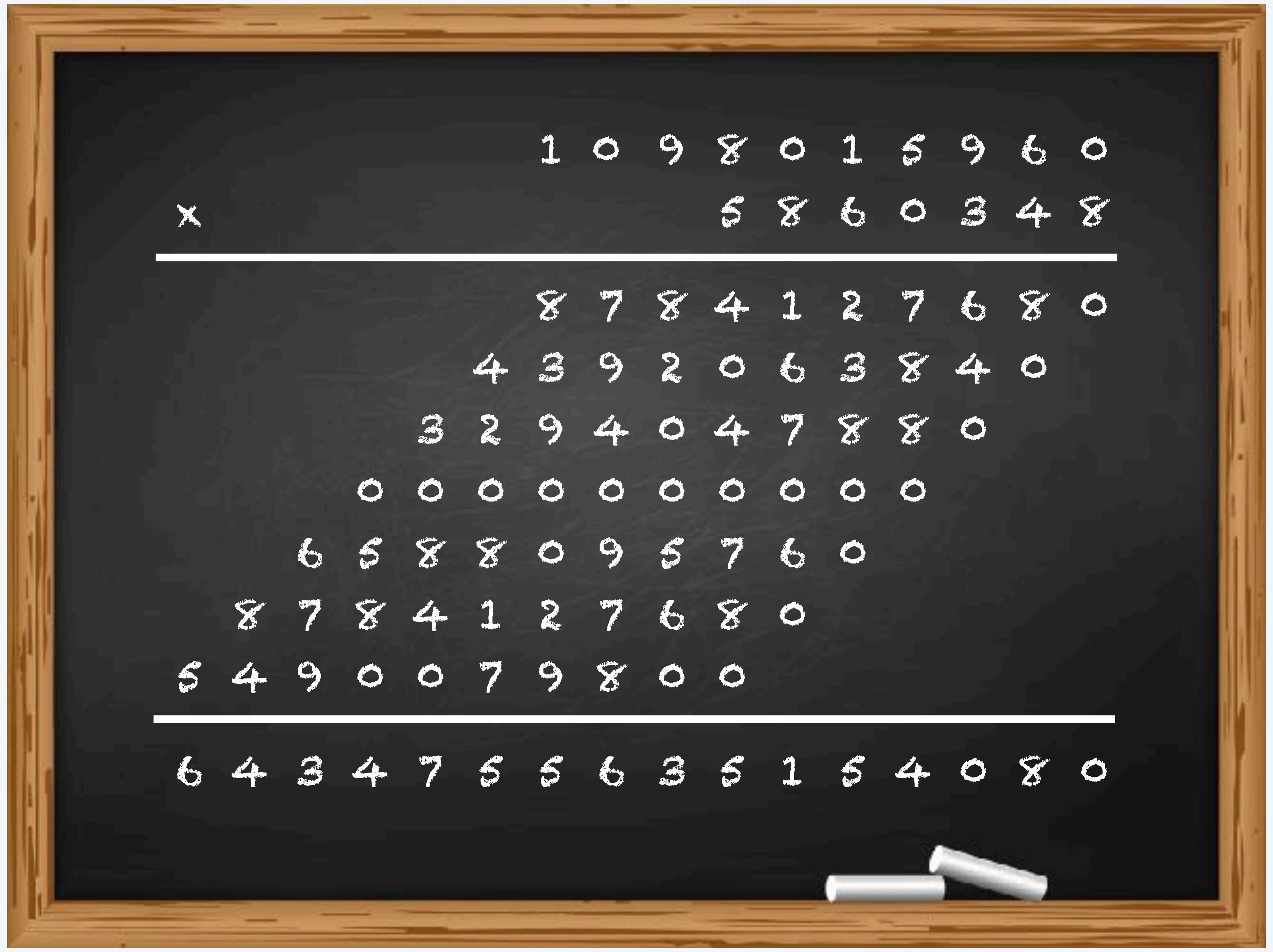
Implemented in scientific computing libraries.

Faster starting at 10,000–100,000 digits.

γ-cruncher: computed 202 trillion (!) digits of π

Java's BigInteger uses efficient multiplication (but not SS).

Lots and lots of clever algorithms!

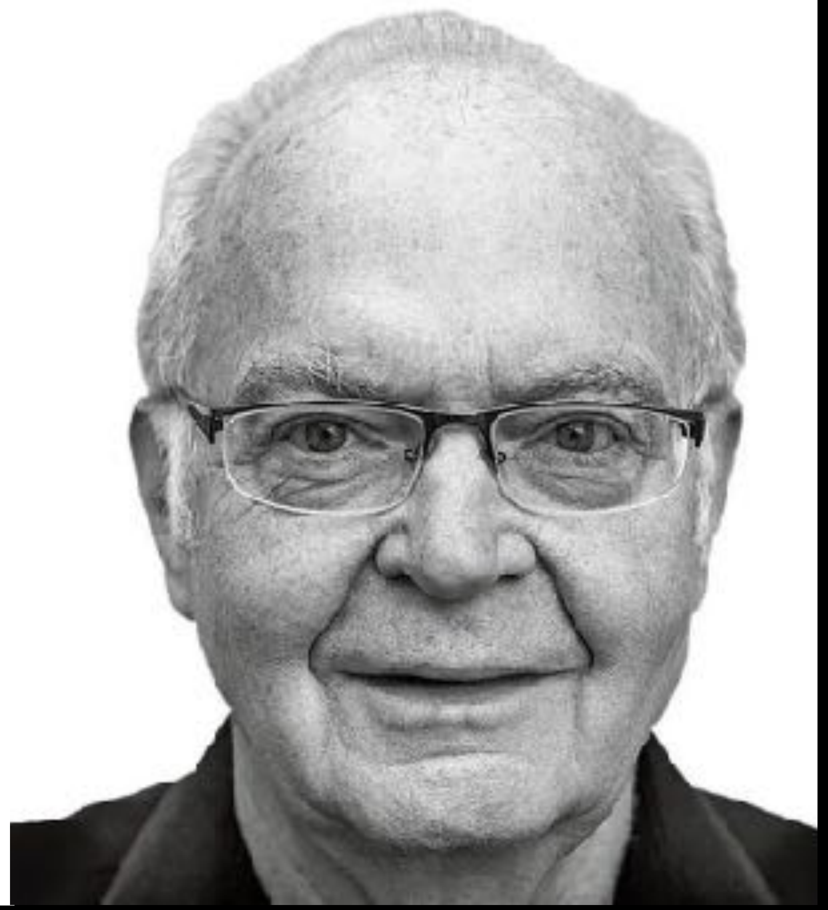


Algorithm	Runtime
Grade school	n^2
Karatsuba	$n^{1.59}$
Toom–Cooke	$n^{1.46}$
Schönhage–Strassen	$n \log n \operatorname{loglog} n$
Harvey–van der Hoeven	$n \log n$

A final thought

“ The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming. ”

— Donald Knuth



Credits

media	source	license
Router	Adobe Stock	Education License
Fourier Transform Diagram	*TikZ.net*	
Blackboard	Adobe Stock	Education License
Donald Knuth	IEEE Computer Society	