



<https://algs4.cs.princeton.edu>

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

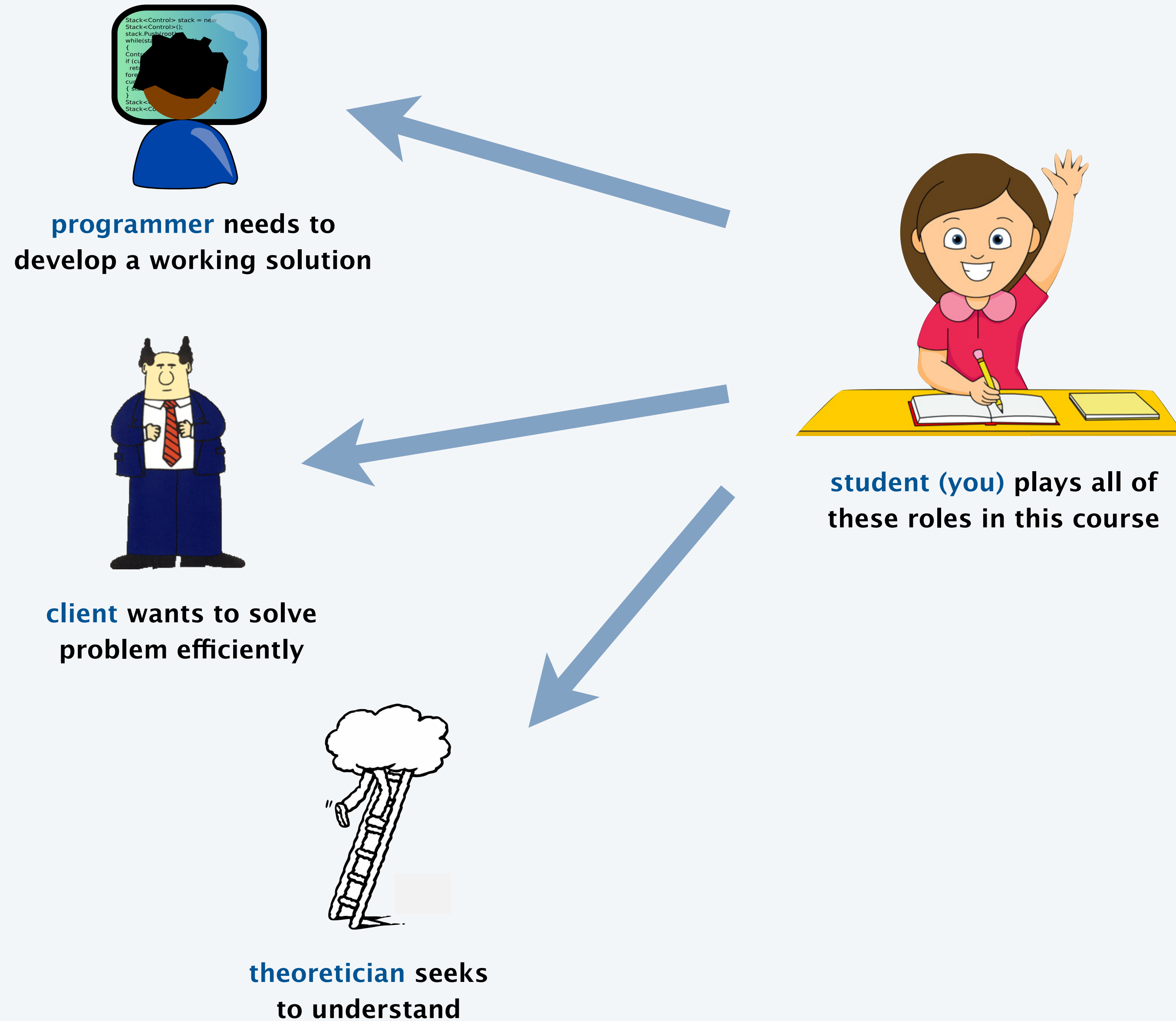


<https://algs4.cs.princeton.edu>

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

Different viewpoints

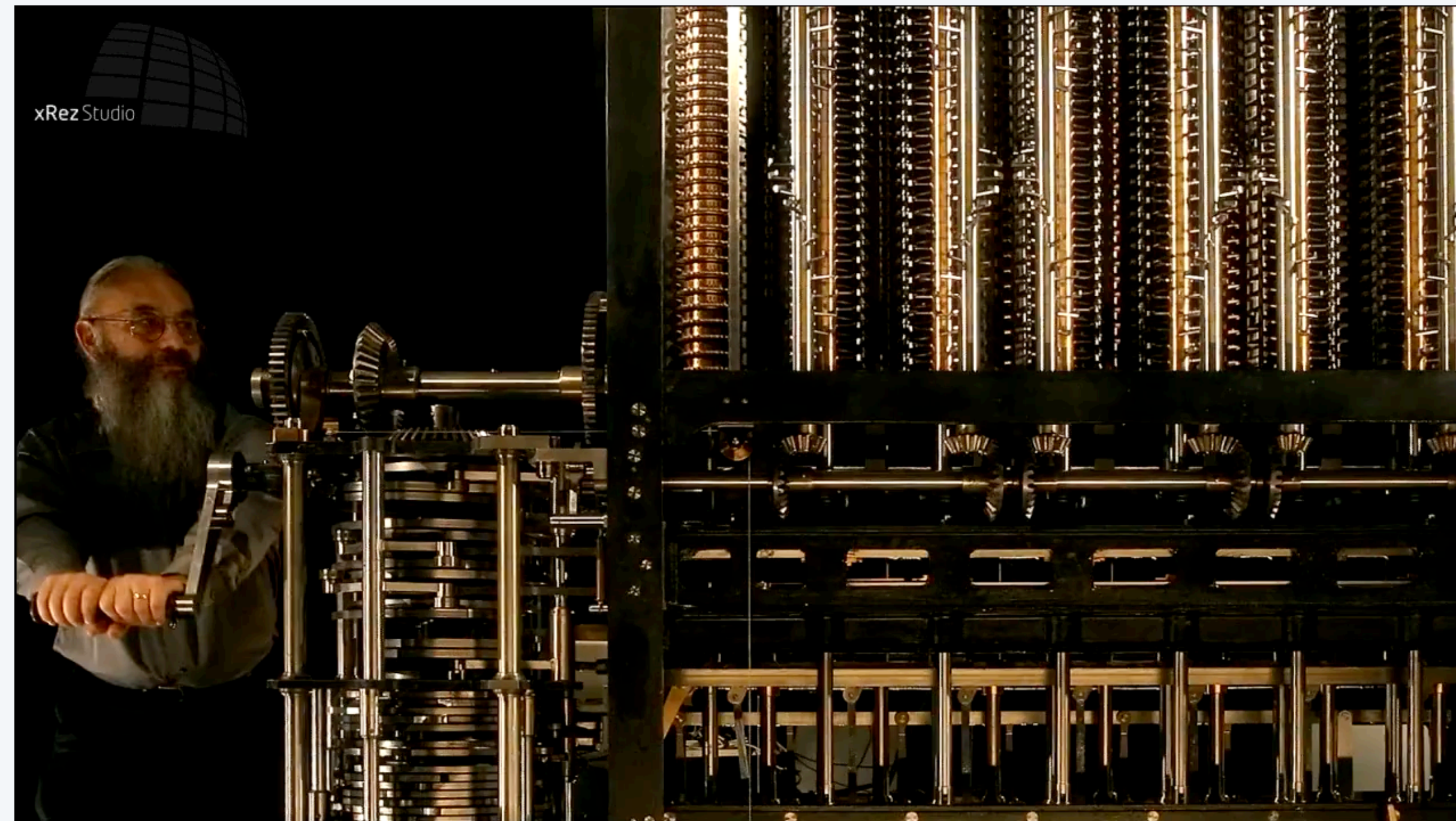


Running time

*“ As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the **shortest time** ? ”* — Charles Babbage (1864)



*how many times
do you have to turn
the crank?*



<https://vimeo.com/49080293>

Running time

“ As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the *shortest time* ? ” — Charles Babbage (1864)

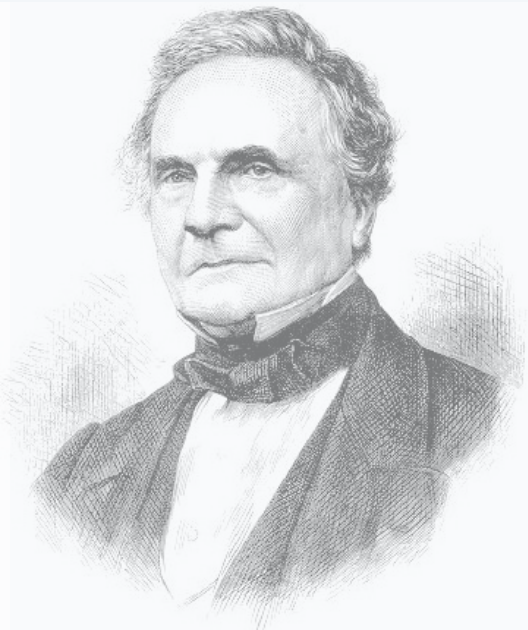
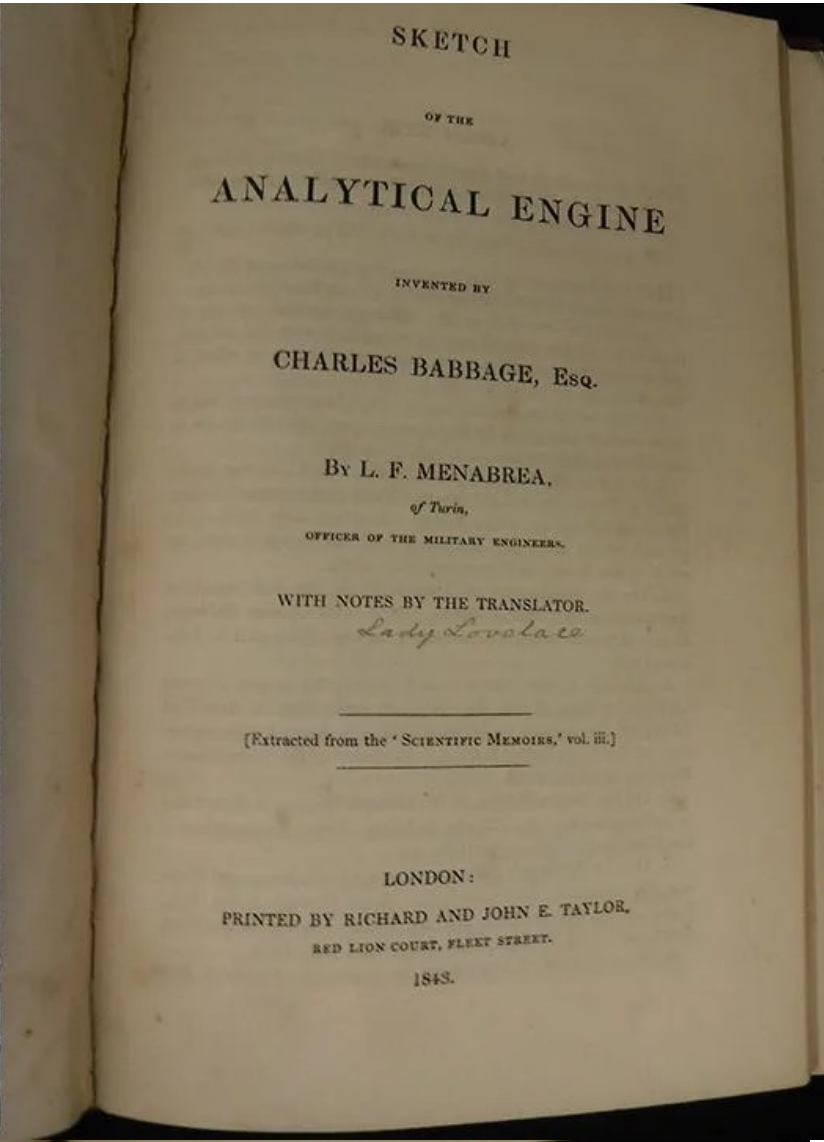


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

Number of Operations. Nature of Operations.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data.										Working Variables.												Result Variables.			
					V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}	V_{14}	V_{15}	V_{16}	V_{17}	V_{18}	V_{19}	V_{20}	V_{21}	V_{22}	V_{23}	V_{24}	V_{25}	
					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
					1	2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
					1	2	n																							
1	\times	$V_2 \times V_1$	V_2	$V_2 = V_2$	$= 2n$	2	n	2n	2n	2n																				
2	$-$	$V_2 - V_1$	V_2	$V_2 = V_2$	$= 2n - 1$	1	...	2n - 1																						
3	$+$	$V_2 + V_1$	V_2	$V_2 = V_2$	$= 2n + 1$	1	...	2n + 1																						
4	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$			0	0																					
5	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	2																								
6	$-$	$V_2 - V_1$	V_{11}	$V_2 = V_2$	$= 2n - 1$																									
7	$-$	$V_2 - V_1$	V_{11}	$V_2 = V_2$	$= 2n - 1$	1	...	n																						
8	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	2																								
9	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$																									
10	\times	$V_2 \times V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$																									
11	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$																									
12	$-$	$V_2 - V_1$	V_{11}	$V_2 = V_2$	$= 2n - 1$	1	...																							
13	$-$	$V_2 - V_1$	V_{11}	$V_2 = V_2$	$= 2n - 1$	1	...																							
14	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
15	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
16	\times	$V_2 \times V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
17	$-$	$V_2 - V_1$	V_{11}	$V_2 = V_2$	$= 2n - 1$	1	...																							
18	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
19	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
20	\times	$V_2 \times V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
21	\times	$V_2 \times V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
22	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
23	$-$	$V_2 - V_1$	V_{11}	$V_2 = V_2$	$= 2n - 1$	1	...																							
Here follows a repetition of Operations thirteen to twenty-three.																														
24	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							
25	$+$	$V_2 + V_1$	V_{11}	$V_2 = V_2$	$= 2n + 1$	1	...																							

Ada Lovelace’s algorithm to compute Bernoulli numbers on Analytical Engine (1843)



An algorithmic success story

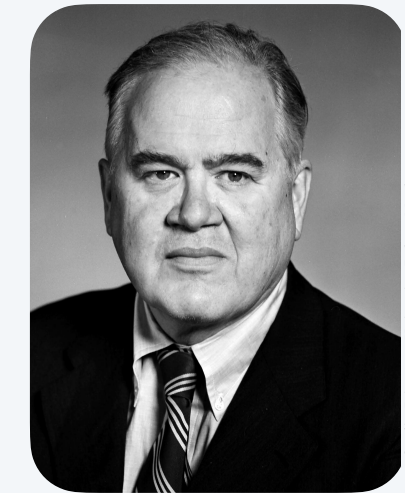
Goal. Multiply two univariate polynomials of degree n .

$$(x^3 + x^2 - 2x + 1) \cdot (3x^3 - x^2 + 2x + 1) = 3x^6 + 2x^5 - 5x^4 + 8x^3 - 4x^2 + 1$$

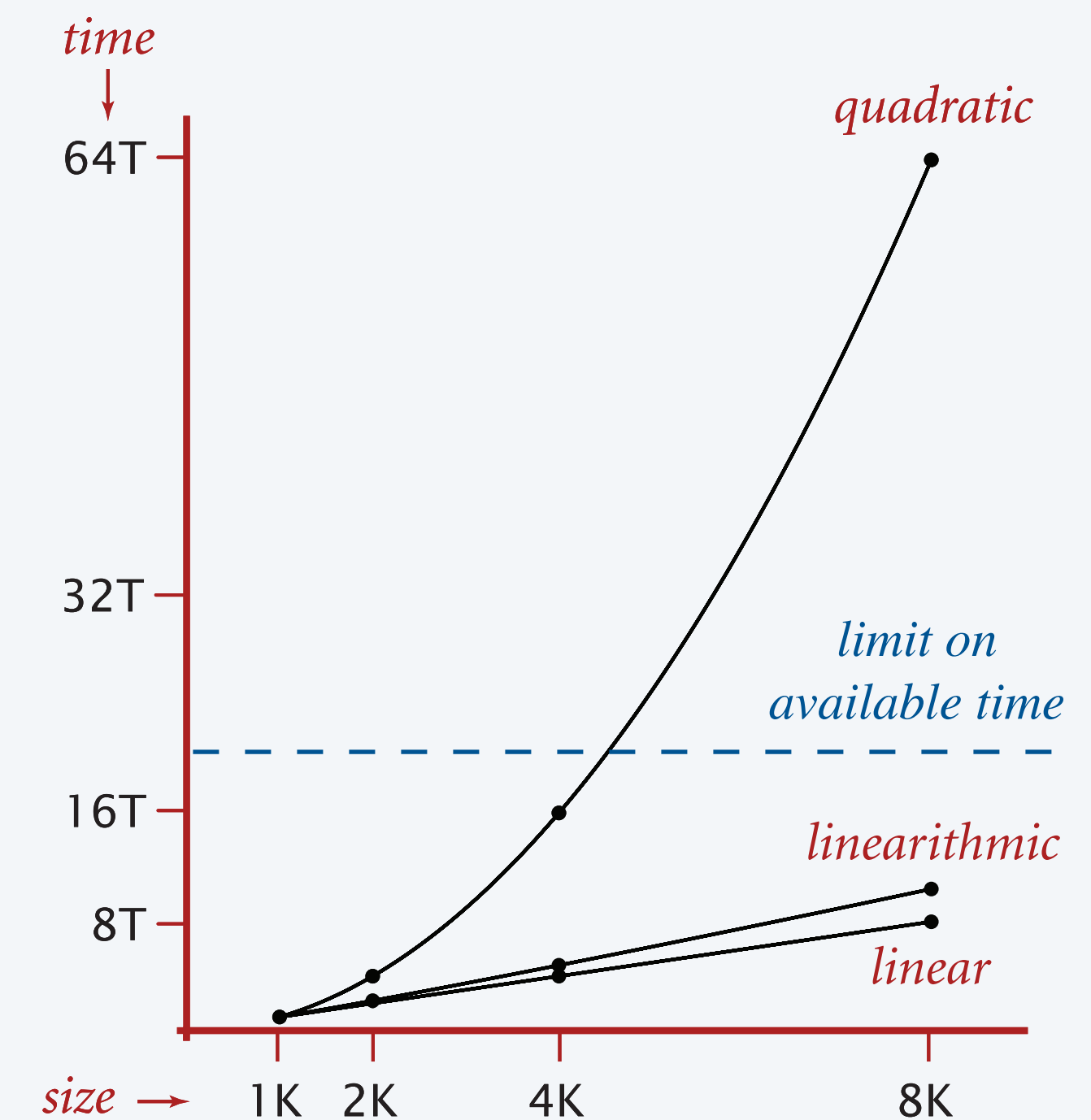
- Applications: JPEG compression, MRI, astrophysics,
- Grade-school algorithm: $\Theta(n^2)$ steps.
- FFT algorithm: $\Theta(n \log n)$ steps, **enables new technology**.



James
Cooley



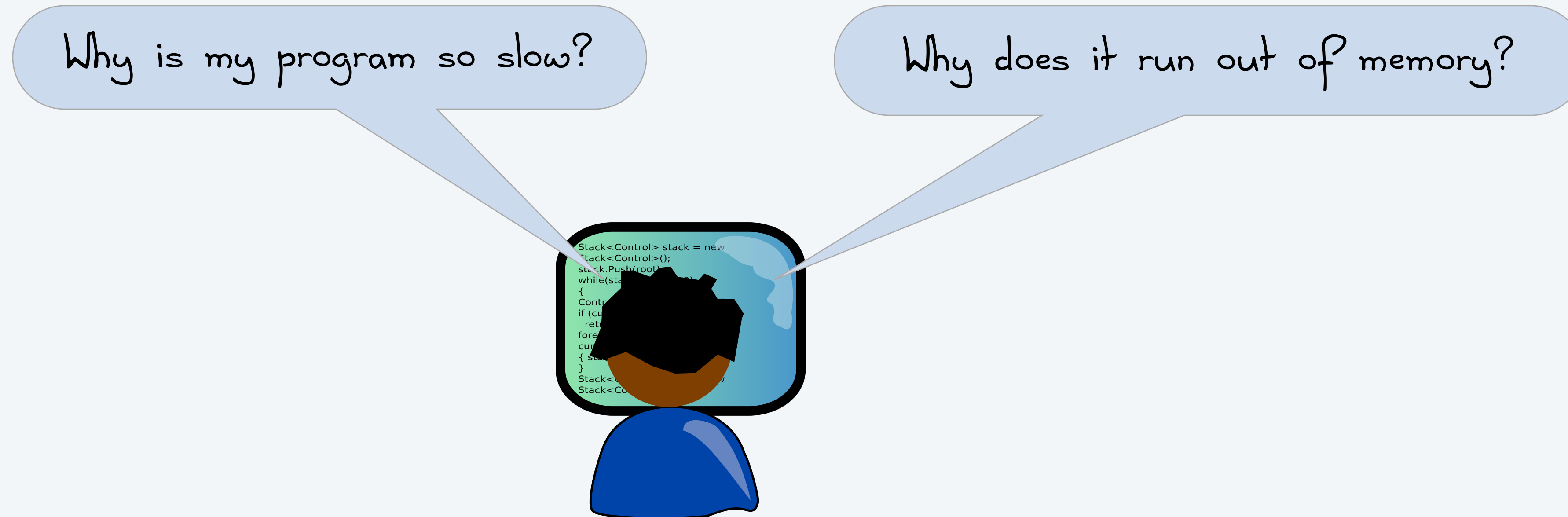
John
Tukey



The challenge

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?



Our approach. Combination of **experiments** and **mathematical modeling**.

Example: 3-SUM



3-SUM. Given n distinct integers, how many triples sum to exactly zero?

```
~/cos226/analysis> more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

~/cos226/analysis> java ThreeSum 8ints.txt
4
```

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

Context. Connected with problems in computational geometry (and computer games!).

Open problem! What is the running time of the optimal algorithm for 3-SUM ?



3-SUM: brute-force algorithm

```
public class ThreeSum {
```

```
    public static int count(int[] a) {
```

```
        int n = a.length;
```

```
        int count = 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)
```

```
                        count++;
```

```
        return count;
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        In in = new In(args[0]);
```

```
        int[] a = in.readAllInts();
```

```
        StdOut.println(count(a));
```

```
    }
```

```
}
```

← *count distinct triples*

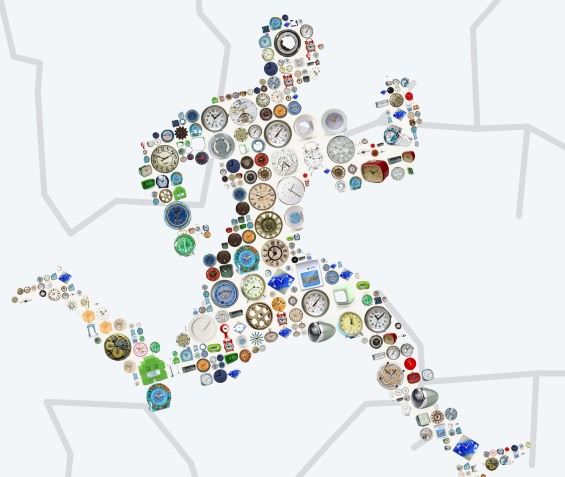
← *for simplicity,
ignore integer overflow*



<https://algs4.cs.princeton.edu>

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

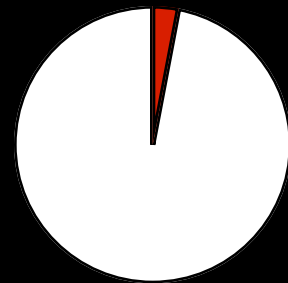


Measuring the running time

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

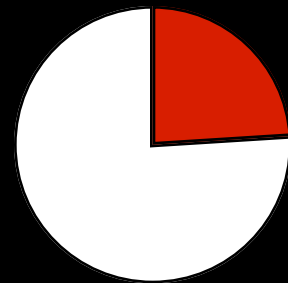
Observation. The running time $T(n)$ increases as a function of the input size n .

```
~/cos226/analysis> java ThreeSum 1Kints.txt  
70
```



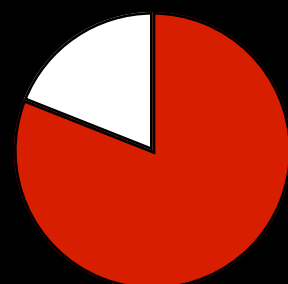
tick tick

```
~/cos226/analysis> java ThreeSum 2Kints.txt  
528
```



*tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick*

```
~/cos226/analysis> java ThreeSum 3Kints.txt  
1670
```



*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 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) †
1,000	0.21
1,500	0.71
2,000	1.63
2,500	3.11
3,000	5.43
4,000	12.8
5,000	25.0
7,500	84.4
10,000	199.3

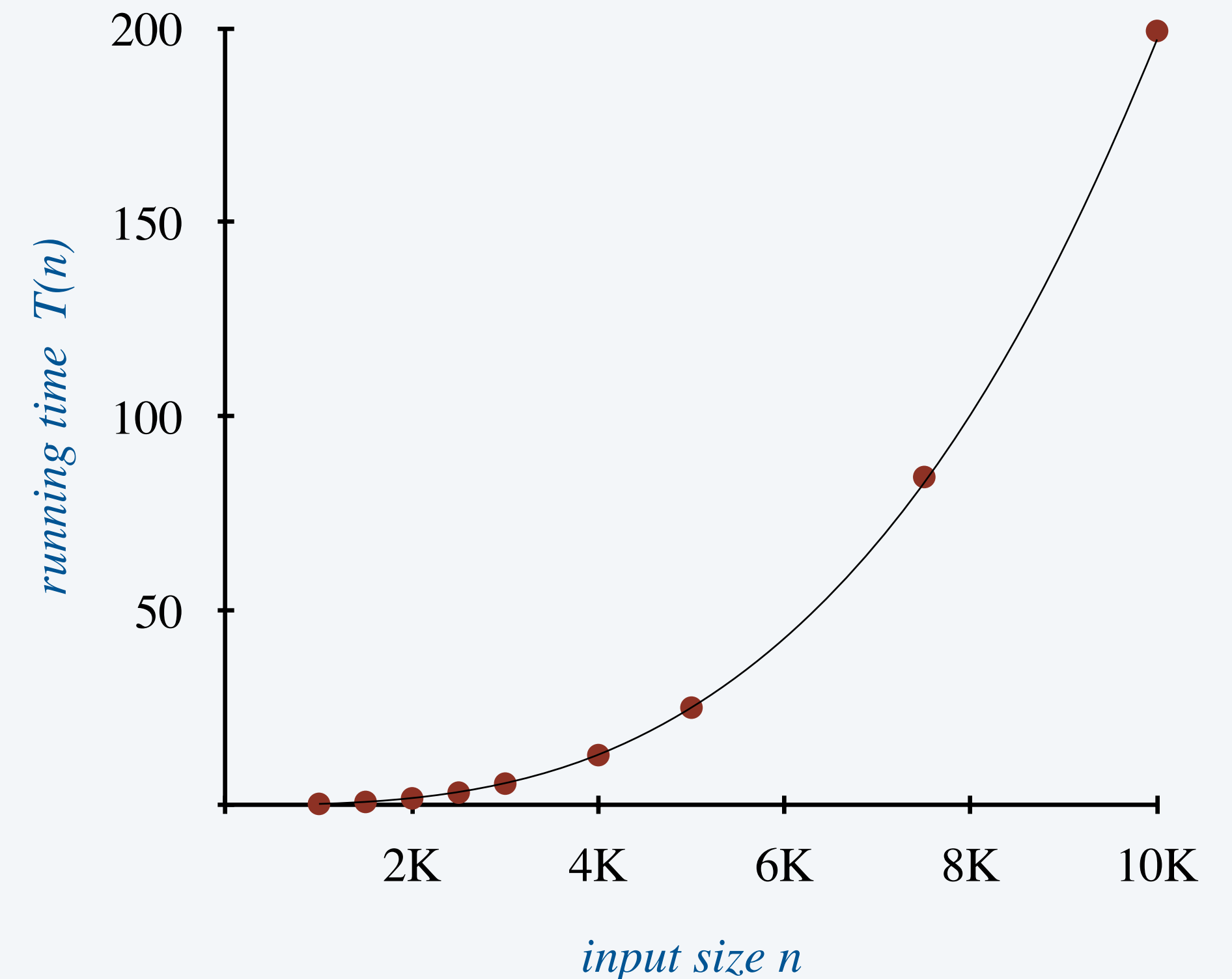


† Apple M2 Pro with 32 GB memory
running OpenJDK 11 on MacOS Ventura

Data analysis: standard plot

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

n	time (seconds) †
1,000	0.21
1,500	0.71
2,000	1.63
2,500	3.11
3,000	5.43
4,000	12.8
5,000	25.0
7,500	84.4
10,000	199.3



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 ?

Doubling test: estimating the exponent b

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

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

n	time (seconds)	ratio	log ₂ ratio
500	0.05	—	—
1,000	0.21	4.20	2.07
2,000	1.63	7.76	2.96
4,000	12.8	7.85	2.97
8,000	103.1	8.05	3.01
16,000	819.0	7.94	2.99

↑
seems to converge to a constant $b \approx 3.0$

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

why the log₂ ratio works

Doubling test: estimating the leading coefficient a

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

- Assume running time satisfies $T(n) = a \times n^b$.
- 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	
500	0.05	—	—	
1,000	0.21	4.20	2.07	
2,000	1.63	7.76	2.96	
4,000	12.8	7.85	2.97	
8,000	103.1	8.05	3.01	
16,000	819.0	7.94	2.99	$819.0 = a \times 16,000^3 \Rightarrow a = 2.00 \times 10^{-10}$

Hypothesis. Running time is about $2.00 \times 10^{-10} \times n^3$ seconds.



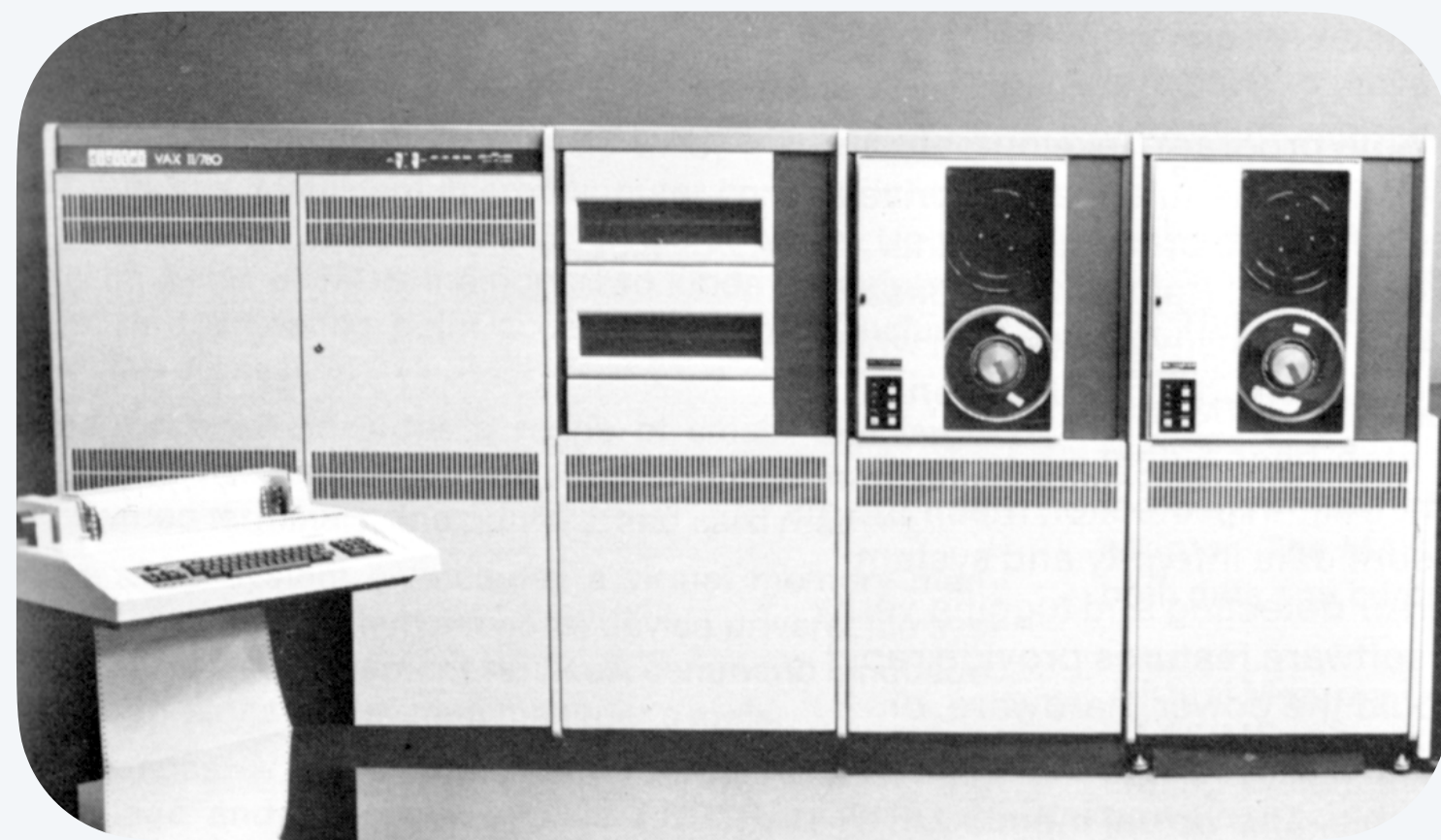
Estimate the running time to solve a problem of size $n = 96,000$.

	n	time (seconds)
A.	39 seconds	
	1,000	0.02
B.	52 seconds	
	2,000	0.05
C.	117 seconds	
	4,000	0.20
D.	350 seconds	
	8,000	0.81
	16,000	3.25
	32,000	13.01

Order of growth

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

Note. That factor can be several orders of magnitude.



1970s
(VAX-11/780)

10,000× *faster*

A large, solid blue arrow pointing from the 1970s computer on the left to the 2020s laptop on the right, indicating a significant performance improvement.

2020s
(Macbook Pro M2)

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.



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://algs4.cs.princeton.edu>

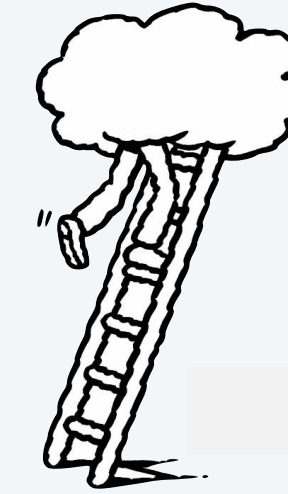
1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *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, ...



Warning. No general-purpose method (e.g., halting problem).

Example: 1-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	<div><div>in practice, depends on caching, bounds checking, ... (see COS 217)</div><div>tedious to count exactly</div></div>
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 $2n$	

† 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) ← “inner loop”
        count++;
```

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>cost model = array accesses</i>
<i>increment</i>	1/10	n to $2\ n$

Simplification 2: asymptotic notations

Tilde notation. Discard lower-order terms.

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

← rigorous definitions involve limits

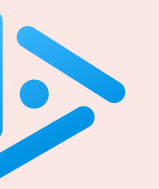
function	tilde notation	big Theta ← “order of growth”
$4 n^5 + 20 n^3 + 16$	$\sim 4 n^5$	$\Theta(n^5)$
$0.01 n^2 + 100 n^{4/3} - 56$	$\sim 0.01 n^2$	$\Theta(n^2)$
$2^n + n^5$	$\sim 2^n$	$\Theta(2^n)$
$\underbrace{\frac{1}{6} n^3 - \frac{1}{2} n^2 + \frac{1}{3} n}_{\text{discard lower-order terms}}$	$\sim \frac{1}{6} n^3$	$\Theta(n^3)$

discard lower-order terms

(e.g., $n = 1,000$: 166.667 million vs. 166.167 million)

Rationale.

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



Which of the following correctly describes the function $f(n) = n \log_2 n + 3n^2 + 10n$?

- A. $\sim 10n$
- B. $\sim n \log_2 n$
- C. $\sim n^2$
- D. $\Theta(n \log n)$
- E. $\Theta(n^2)$

Example: two-sum

Q. Approximately how many operations 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++;
```

← “inner loop”

$$\begin{array}{ccccccccc} i=0 & & i=1 & & i=2 & & & & i=n-2 & & i=n-1 \\ j=1, \dots, n-1 & & j=2, \dots, n-1 & & j=3, \dots, n-1 & & & & j=n-1 & & \text{no } j \\ \downarrow & & \downarrow & & \downarrow & & & & \downarrow & & \downarrow \\ (n-1) & + & (n-2) & + & (n-3) & + & \dots & + & 1 & + & 0 \\ \underbrace{\hspace{15em}} & & & & & & & & & & \\ & & & & & & = & \frac{n(n-1)}{2} \end{array}$$

Step 1. Choose cost model = array accesses.

Step 2. Count number of array accesses $= 2 \times \frac{n(n-1)}{2} \sim 1n^2$.

↑
inner loop makes
2 array accesses

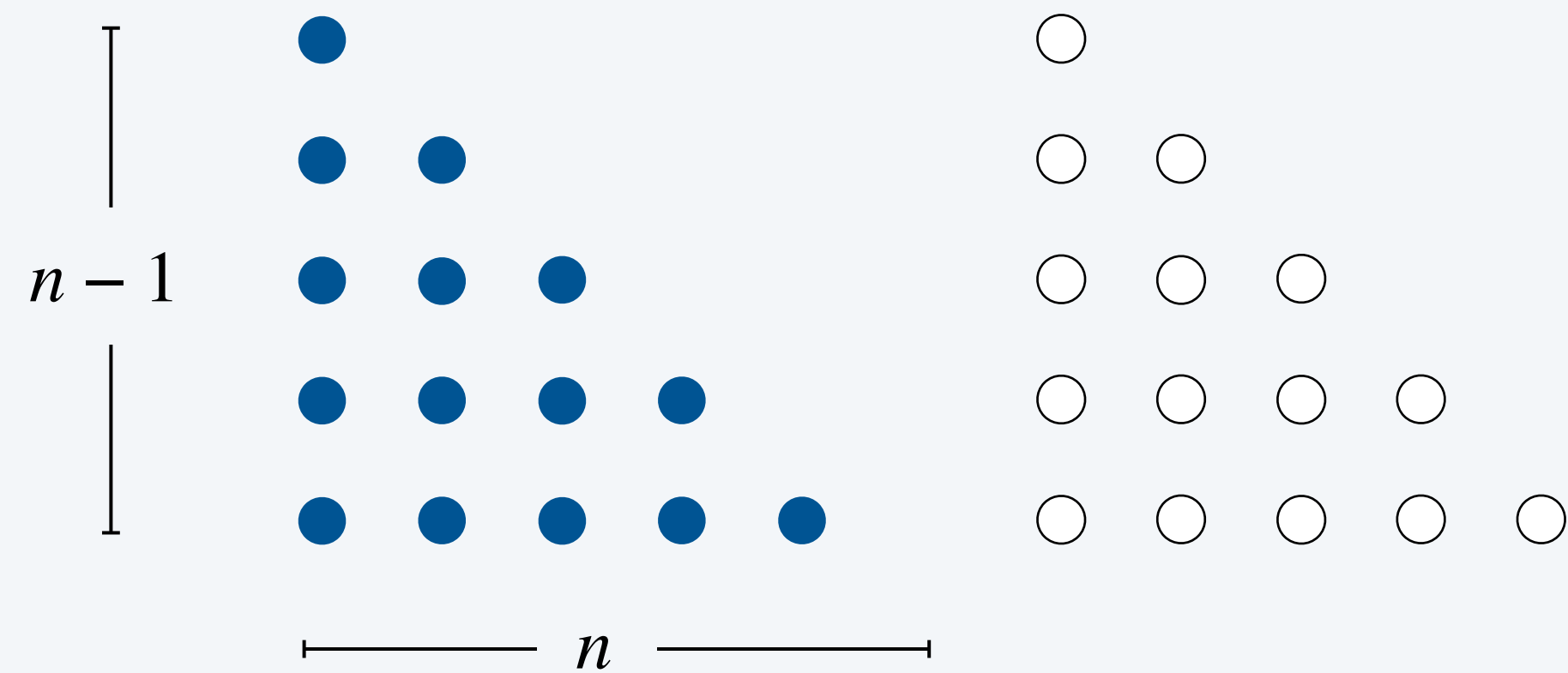
Nested loops.

- If loops are independent, analyze separately and multiply.
- If loops are dependent, write a sum (and simplify).

Triangular sum

Claim. $0 + 1 + \dots + (n-2) + (n-1) = \frac{1}{2} n(n-1).$

Proof. [by picture]



$$2 \times (1 + 2 + 3 + \dots + n-1) = n(n-1)$$

$$\Rightarrow (1 + 2 + 3 + \dots + n-1) = \frac{1}{2} n(n-1)$$

Example: 3-SUM

Q. Approximately many operations 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++)
        for (int k = j+1; k < n; k++)
            if (a[i] + a[j] + a[k] == 0)
                count++;
```

← “inner loop” $\binom{n}{3} = \frac{n(n-1)(n-2)}{3!} \sim \frac{1}{6}n^3$

see COS 240

Step 1. Choose cost model = array accesses.

Step 2. Count number of array accesses = $\Theta(n^3)$.

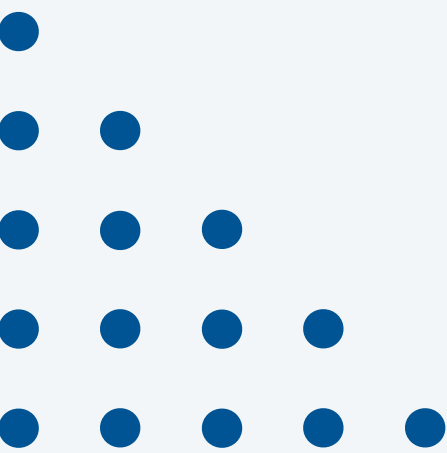
Bottom line. Use **cost model** and **asymptotic notation** to simplify analysis.

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 > 0; 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</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

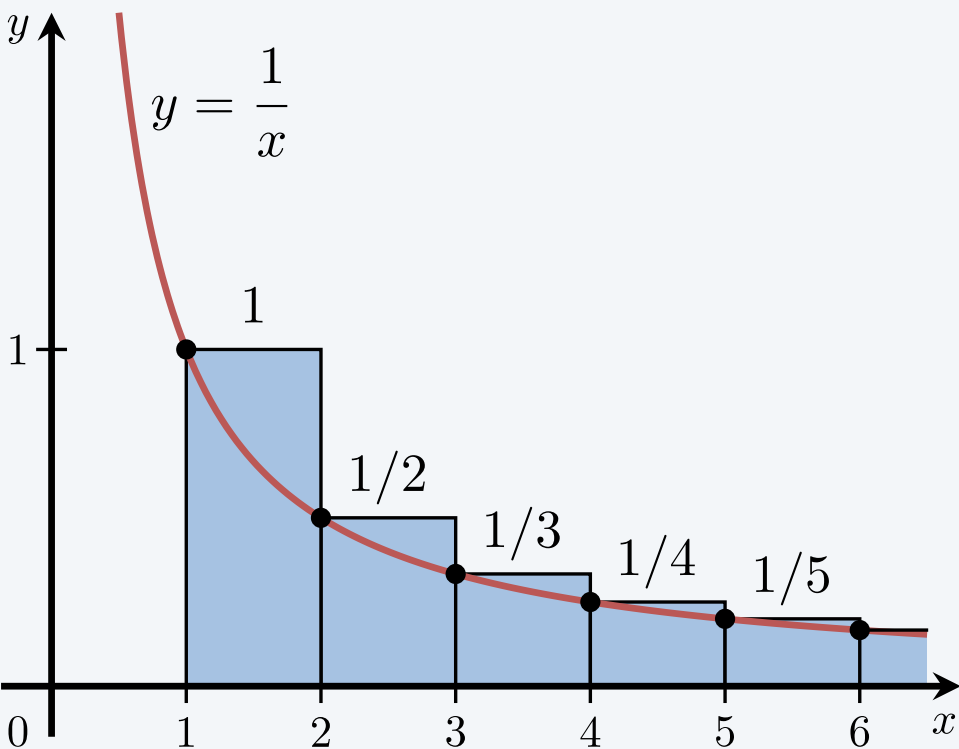
Some useful discrete sums and approximations

Triangular sum. $1 + 2 + 3 + \dots + n \sim \frac{1}{2} n^2$



Logarithmic sum. $\log_2 1 + \log_2 2 + \log_2 3 + \dots + \log_2 n \sim \int_{x=1}^n \log_2 x \, dx = n \log_2 n$

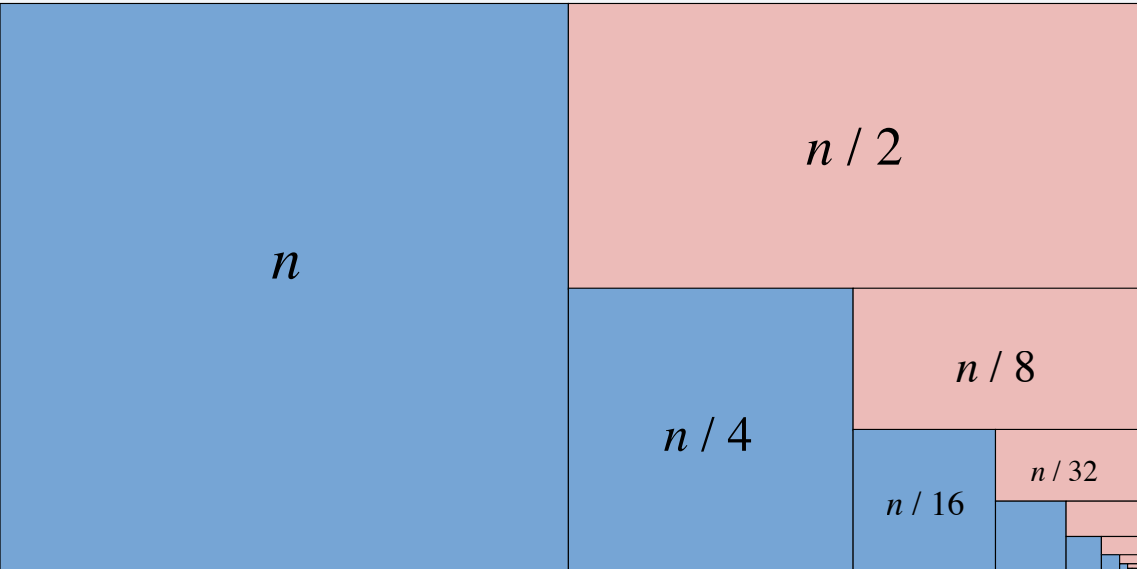
Harmonic sum. $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \sim \int_{x=1}^n \frac{1}{x} \, dx = \ln n$



Geometric sum. $1 + 2 + 4 + 8 + \dots + n = 2n - 1$

Geometric sum'. $n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n - 1$

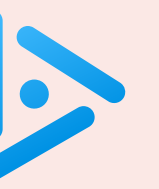
n a power of 2



Geometric sum meme



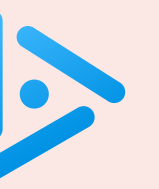
<https://marekbennett.com/2014/03/06/recursive-load>



Approximately how many **array accesses** as a function of n ?

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

- A. $\sim n^2 \log_2 n$
- B. $\sim 3/2 n^2 \log_2 n$
- C. $\sim 1/2 n^3$
- D. $\sim 3/2 n^3$



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

```
int count = 0;
for (int i = n; i >= 1; i = i/2)
    for (int j = 1; j <= i; j++)
        count++;
```

- A. $\Theta(n)$
- B. $\Theta(n \log n)$
- C. $\Theta(n^2)$
- D. $\Theta(2^n)$





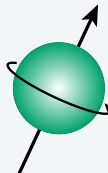



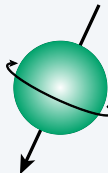

<https://algs4.cs.princeton.edu>

1.4 ANALYSIS OF ALGORITHMS

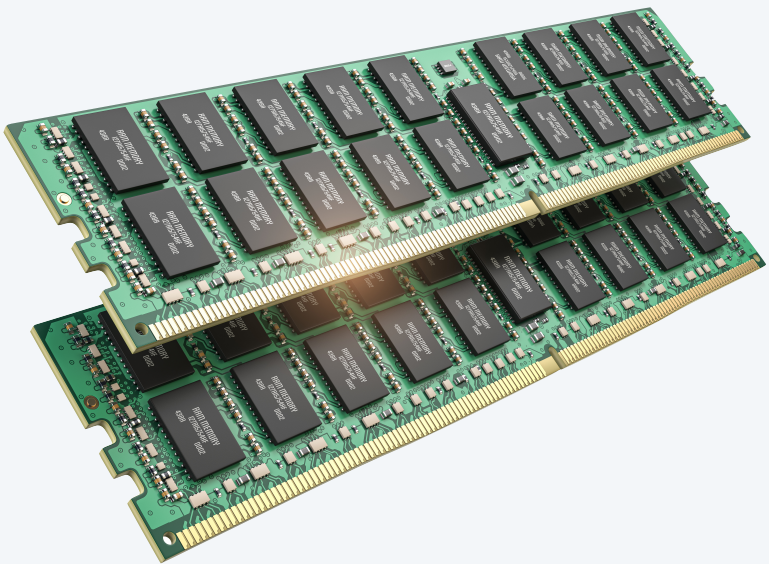
- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

Memory basics

Bit. 0 or 1.

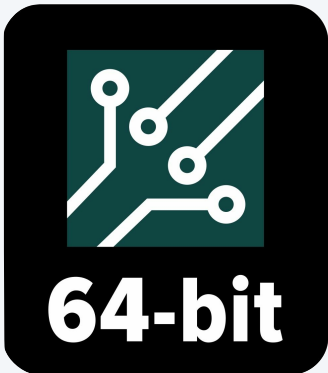
0				
1				

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
(e.g., MB = 2^{20} bytes)*

64-bit machine. We assume a 64-bit machine with 8-byte pointers.



↑
*some JVMs “compress” pointers
to 4 bytes to avoid this cost*

Typical memory usage for primitive types and arrays

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8
primitive types	

type	bytes
boolean[]	$1n + 24$ ← <i>wasteful</i> <i>(but ~ 36n bytes in Python 3)</i>
int[]	$4n + 24$
double[]	$8n + 24$ ← <i>array overhead = 24 bytes</i>
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)	

type	bytes
object reference	8
	↑ <i>64-bit machine</i>

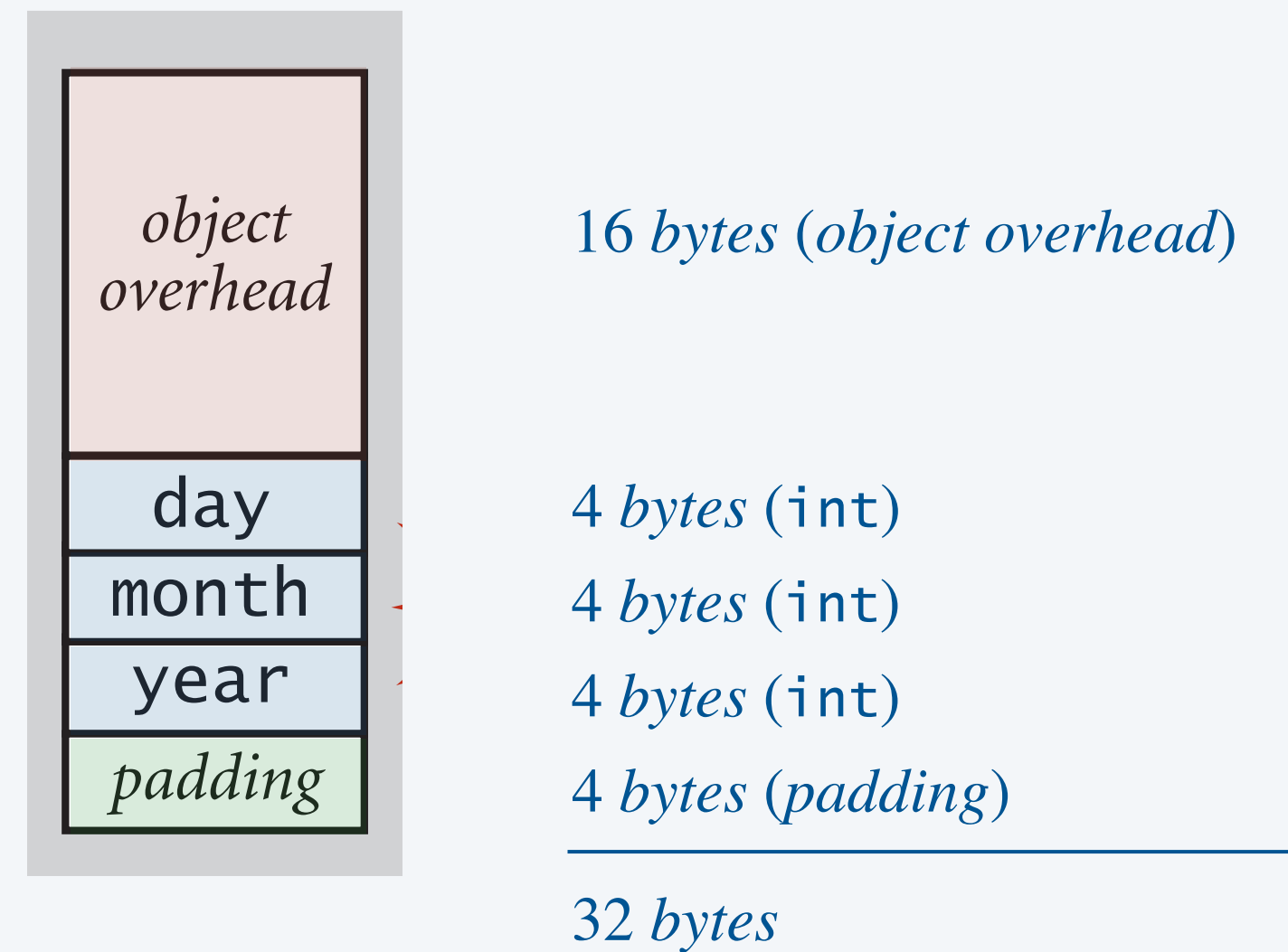
Typical memory usage for objects in Java

Object overhead. 16 bytes.

Padding. Round up memory of each object to be a multiple of 8 bytes.

Ex. Each `Date` object uses 32 bytes of memory.

```
public class Date {  
    private int day;  
    private int month;  
    private int year;  
  
    ...  
}
```





How much memory does a `WeightedQuickUnionUF` object use as a function of n ?

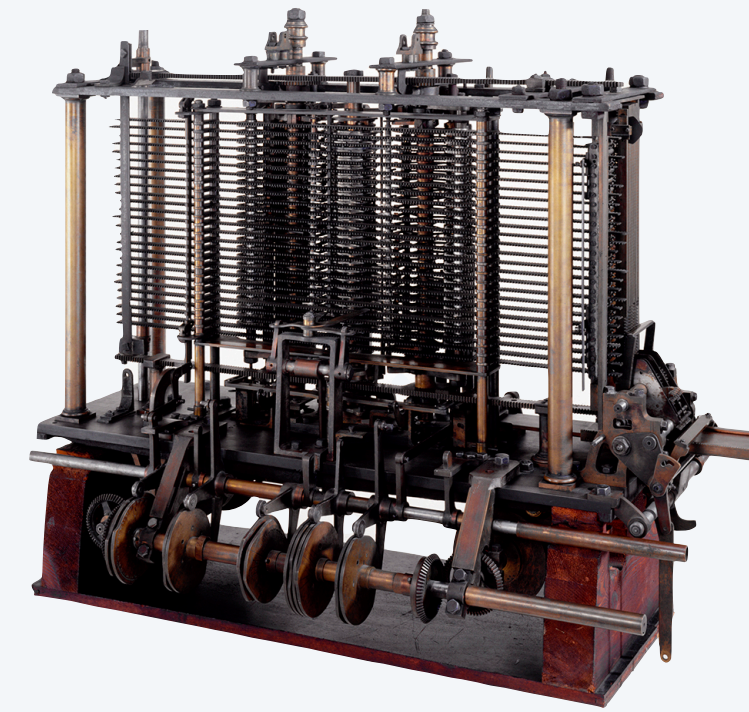
- A. $\sim 4n$ bytes
- B. $\sim 8n$ bytes
- C. $\sim 4n^2$ bytes
- D. $\sim 8n^2$ bytes

```
public class WeightedQuickUnionUF {  
    private int[] parent;  
    private int[] size;  
    private int count;  
  
    public WeightedQuickUnionUF(int n) {  
        parent = new int[n];  
        size = new int[n];  
  
        count = 0;  
        for (int i = 0; i < n; i++)  
            parent[i] = i;  
        for (int i = 0; i < n; i++)  
            size[i] = 1;  
    }  
    ...  
}
```


Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law.
- Formulate a hypothesis for running time.
- Model enables us to **make predictions**.



Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use cost model and asymptotic notations to simplify analysis.
- Model enables us to **explain behavior**.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \sim n$$

This course. Learn to use both.

Credits

image	source	license
<i>Charles Babbage</i>	<u>The Illustrated London News</u>	<u>public domain</u>
<i>Babbage Enginine in Operation</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>Galaxies Colliding</i>	<u>SaltyMikan</u>	
<i>James Cooley</i>	<u>IEEE</u>	
<i>John Tukey</i>	<u>Princeton University</u>	
<i>Programmer Icon</i>	<u>Jaime Botero</u>	<u>public domain</u>
<i>Head in the Clouds</i>	<u>Ellis Nadler</u>	<u>education</u>
<i>Student Raising Hand</i>	<u>classroomclipart.com</u>	<u>educational use</u>
<i>Running Time</i>	<u>pano.si</u>	
<i>Analog Stopwatch</i>	<u>Adobe Stock</u>	<u>education license</u>

Credits

image	source	license
<i>Apple M4 Chip</i>	<u>Apple</u>	
<i>Macbook Pro M2</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>Harmonic Integral</i>	<u>Wikimedia</u>	<u>public domain</u>
<i>Geometric Series</i>	<u>Wikimedia</u>	<u>CC BY-SA 3.0</u>
<i>Recursive Load</i>	<u>Marek Bennett</u>	
<i>The Yoda of Silicon Valley</i>	<u>New York Times</u>	
<i>Babbage's Analytical Engine</i>	<u>Science Museum, London</u>	<u>CC BY-SA 2.0</u>
<i>Alan Turing</i>	<u>Science Museum, London</u>	

A final thought

“It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then give them various weights.” — Alan Turing (1947)

