



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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory (read your textbook)*



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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Cast of characters



Programmer needs to develop a working solution.



Student (you) might play any or all of these roles someday.



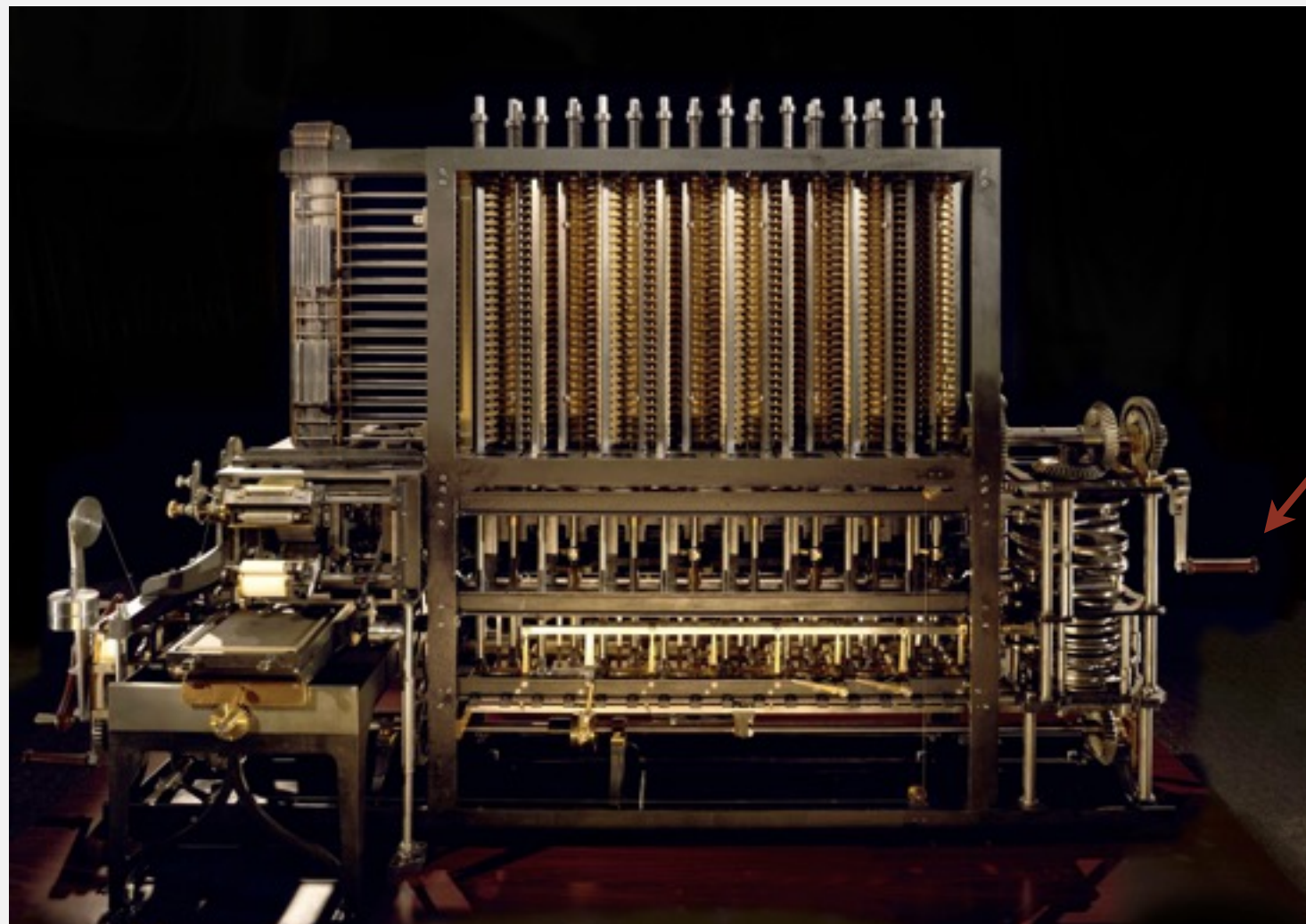
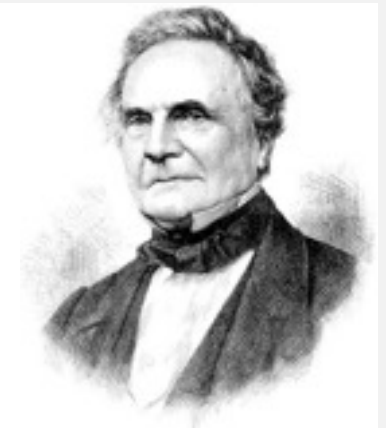
Client wants to solve problem efficiently.



Theoretician seeks to understand.

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?



Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course
(CIS 121)

theory of algorithms
(CIS 320)

Primary practical reason: avoid performance bugs.



**client gets poor performance because programmer
did not understand performance characteristics**



Another algorithmic success story

Discrete Fourier transform.

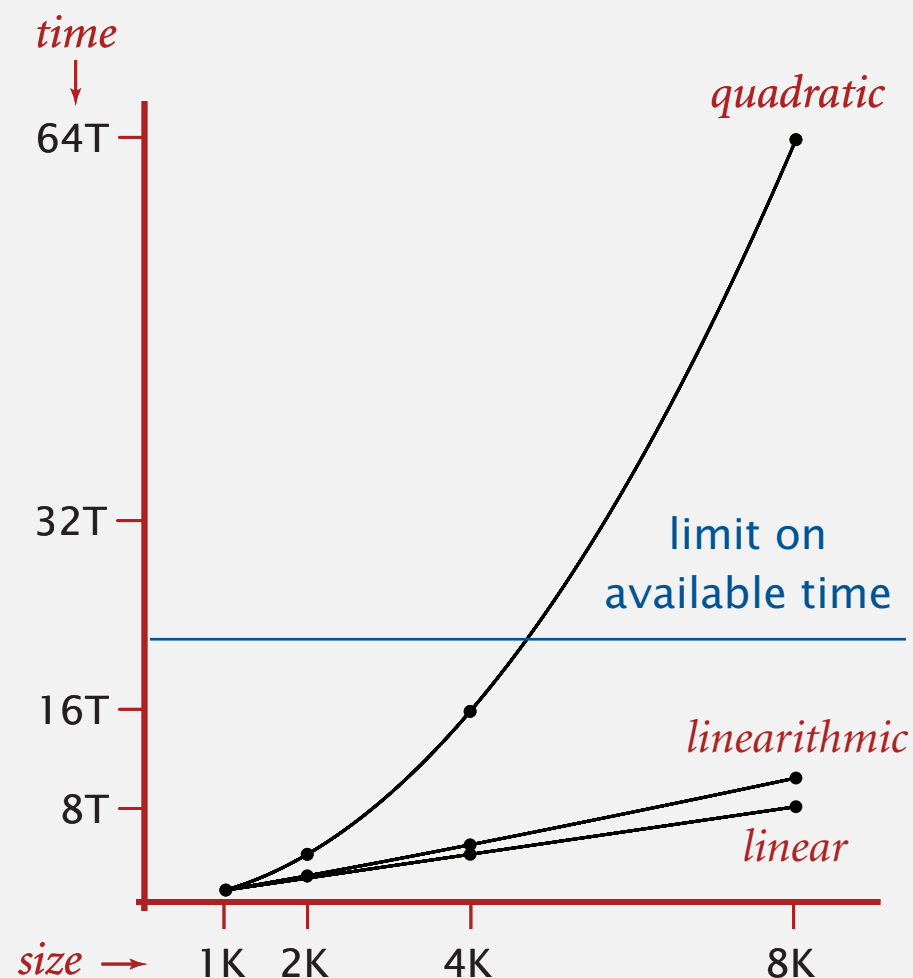
- Express signal as weighted sum of sines and cosines.
- Applications: DVD, JPEG, MRI, astrophysics,
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, **enables new technology.**



James
Cooley



John
Tukey



The challenge

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

Why is my program so slow ?

Why does it run out of memory ?



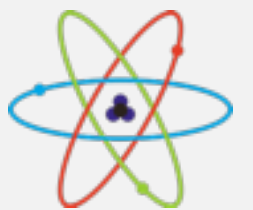
Insight. [Knuth 1970s] Use **scientific method** to understand performance.

Scientific method applied to the analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.

- **Observe** some feature of the natural world.
- **Hypothesize** a model that is consistent with the observations.
- **Predict** events using the hypothesis.
- **Verify** the predictions by making further observations.
- **Validate** by repeating until the hypothesis and observations agree.



Principles.

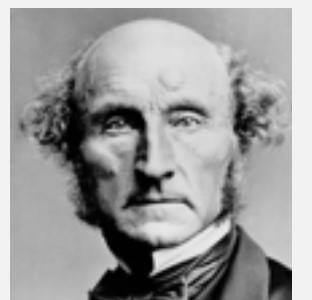
- Experiments must be **reproducible**.
- Hypotheses must be **falsifiable**.



Francis
Bacon



René
Descartes



John Stuart
Mills

Feature of the natural world. Computer itself.



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

1.4 ANALYSIS OF ALGORITHMS

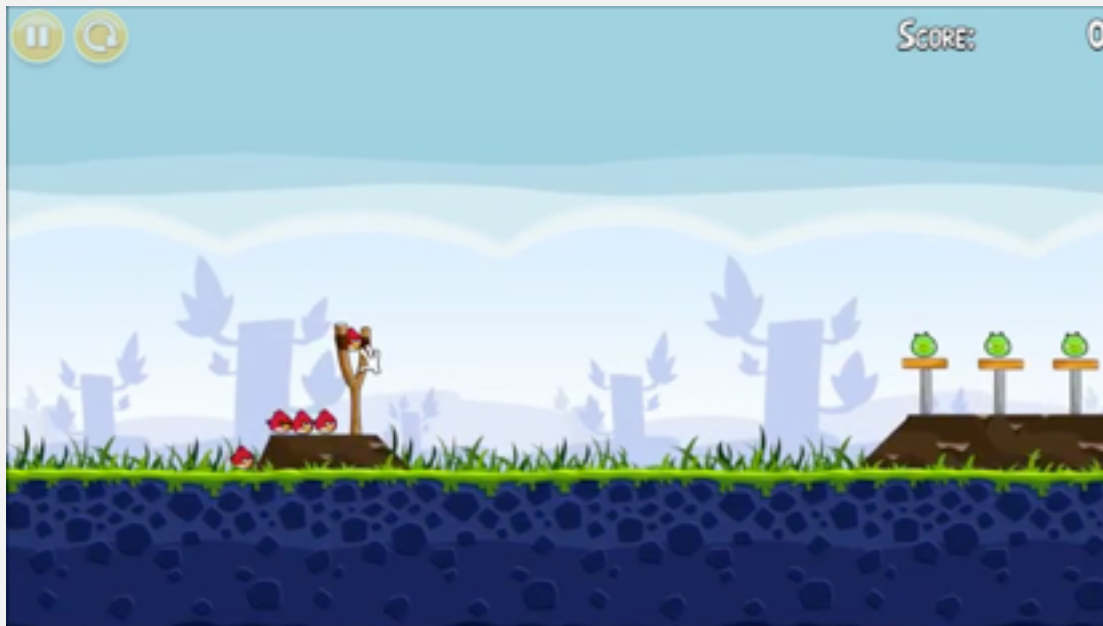
- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Running example: 3-SUM

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

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

% 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. Deeply related to problems in computational geometry.

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) ← check each triple
                        count++;
        return count;
    }

    public static void main(String[] args)
    {
        In in = new In(args[0]);
        int[] a = in.readAllInts();
        StdOut.println(count(a));
    }
}
```

Measuring the running time

Q. How to time a program?

A. Manual.



```
% java ThreeSum 1Kints.txt
```



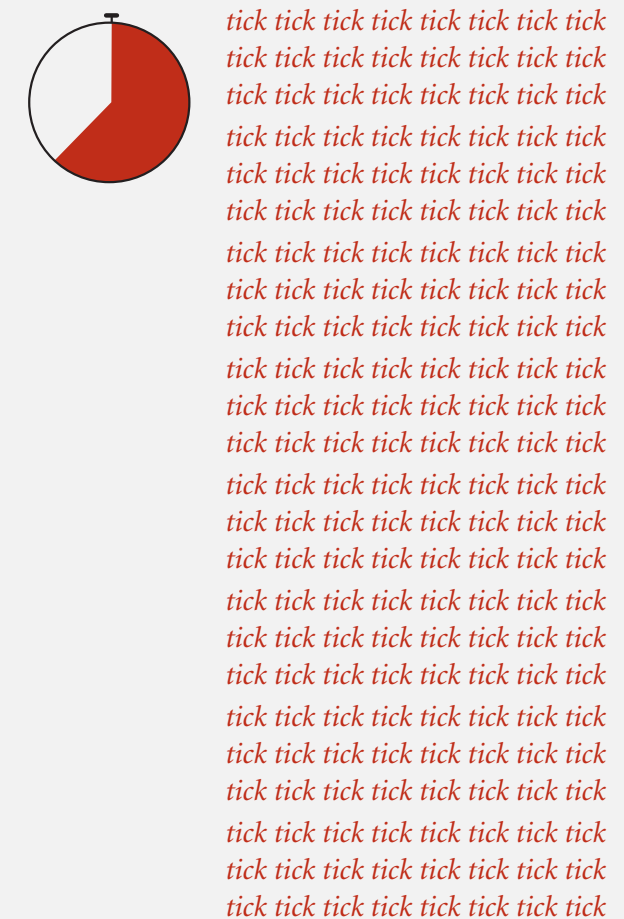
70

```
% java ThreeSum 2Kints.txt
```



528

```
% java ThreeSum 4Kints.txt
```



4039

Measuring the running time

Q. How to time a program?

A. Automatic.

```
public class Stopwatch (part of stdlib.jar)
```

```
    Stopwatch() create a new stopwatch
```

```
    double elapsedTime() time since creation (in seconds)
```

```
public static void main(String[] args)
{
    In in = new In(args[0]);
    int[] a = in.readAllInts();
    Stopwatch stopwatch = new Stopwatch();
    StdOut.println(ThreeSum.count(a));
    double time = stopwatch.elapsedTime();
    StdOut.println("elapsed time = " + time);
}
```


Empirical analysis

Run the program for various input sizes and measure running time.



Empirical analysis

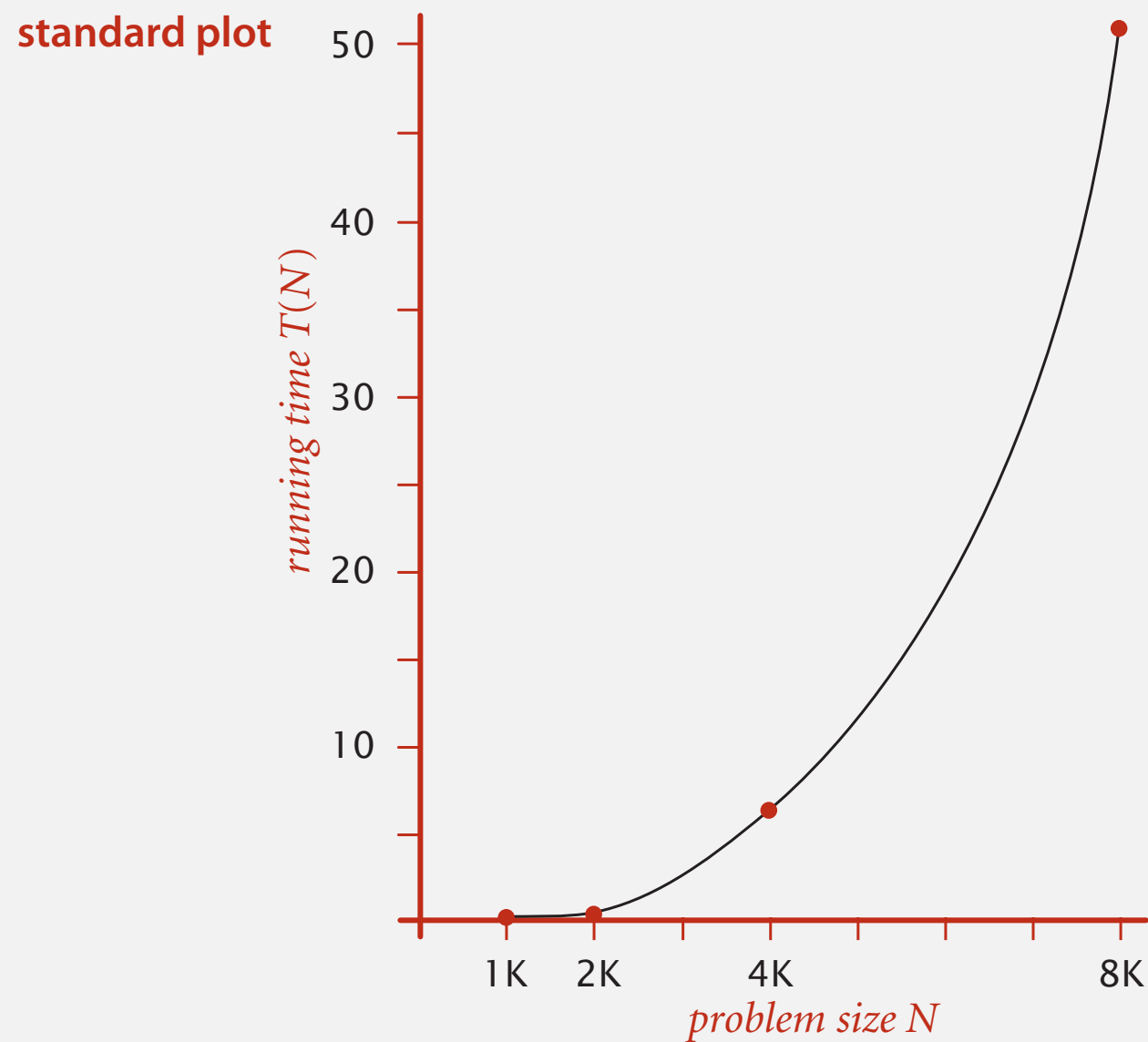
Run the program for various input sizes and measure running time.

N	time (seconds) †
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

† on a 2.8GHz Intel PU-226 with 64GB DDR E3 memory and 32MB L3 cache; running Oracle Java 1.7.0_45-b18 on Springdale Linux v. 6.5

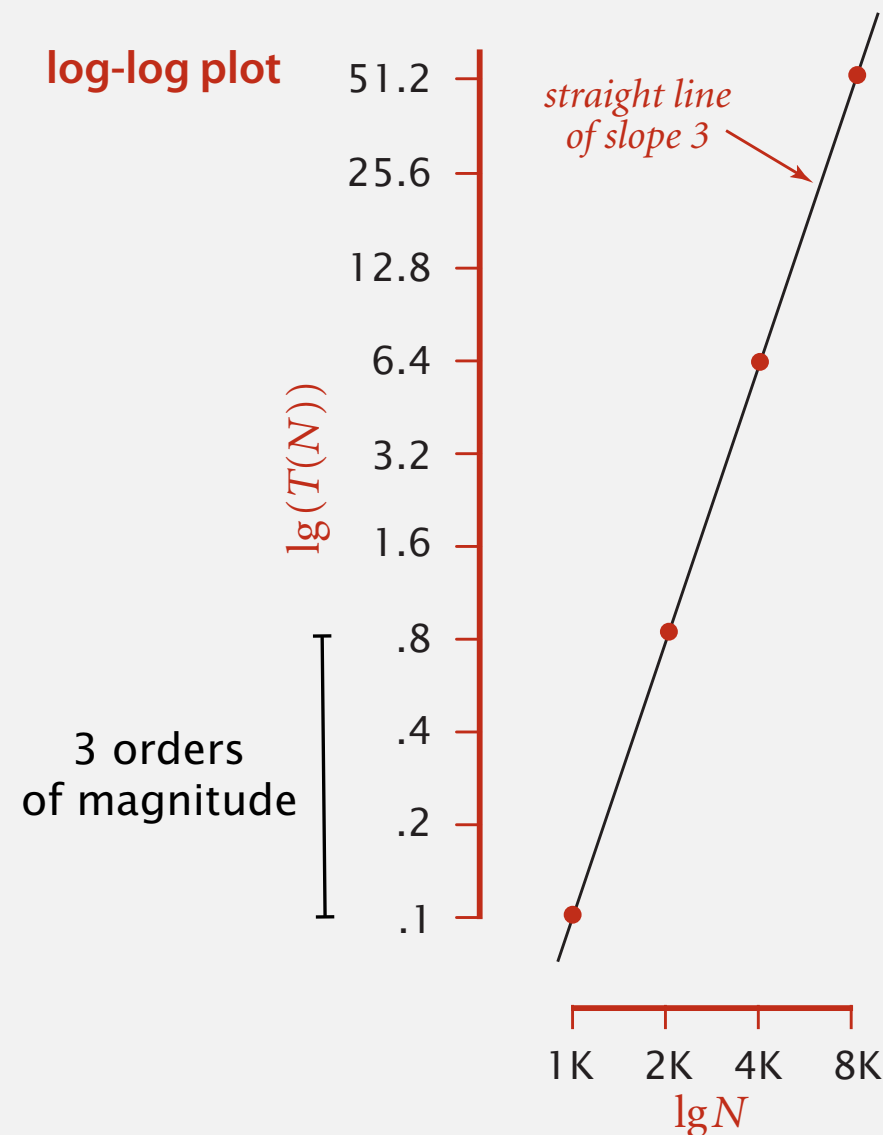
Data analysis

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



Data analysis

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



$$\lg(T(N)) = b \lg N + c$$

$$b = 2.999$$

$$c = -33.2103$$

$$T(N) = a N^b, \text{ where } a = 2^c$$

Regression. Fit straight line through data points: $a N^b$.


Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

power law
slope

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

"order of growth" of running time is about N^3 [stay tuned]



Predictions.

- 51.0 seconds for $N = 8,000$.
- 408.1 seconds for $N = 16,000$.

Observations.

N	time (seconds) †
8,000	51.1
8,000	51
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Run program, **doubling** the size of the input.

N	time (seconds) †	ratio	lg ratio
250	0		–
500	0	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8	3
8,000	51.1	8	3

$$\frac{T(N)}{T(N/2)} = \frac{aN^b}{a(N/2)^b} = 2^b$$

← $\lg(6.4 / 0.8) = 3.0$

↑
seems to converge to a constant $b \approx 3$

Hypothesis. Running time is about $a N^b$ with $b = \lg \text{ratio}$.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Q. How to estimate a (assuming we know b) ?

A. Run the program (for a sufficient large value of N) and solve for a .

N	time (seconds) †
8,000	51.1
8,000	51
8,000	51.1

$$51.1 = a \times 8000^3$$

$$\Rightarrow a = 0.998 \times 10^{-10}$$

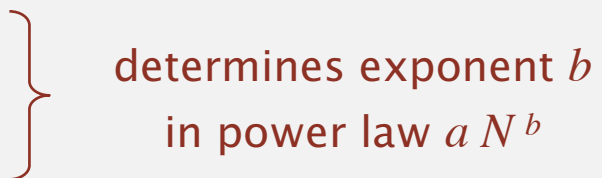
Hypothesis. Running time is about $0.998 \times 10^{-10} \times N^3$ seconds.



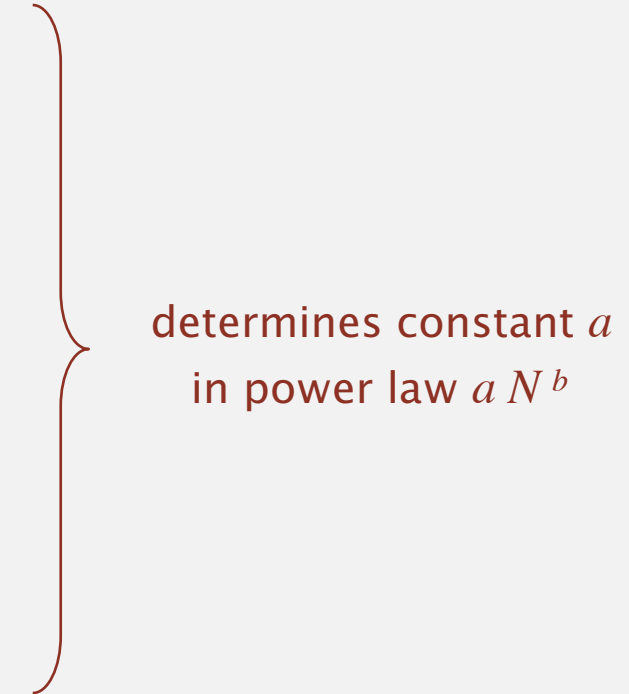
almost identical hypothesis
to one obtained via regression

Experimental algorithmics

System independent effects.

- Algorithm.
 - Input data.
- 
- determines exponent b
in power law $a N^b$

System dependent effects.

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

determines constant a
in power law $a N^b$

Bad news. Sometimes difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

Good news for computer science

Algorithmic experiments are virtually free by comparison with other sciences.



Chemistry
(1 experiment)



Biology
(1 experiment)



Computer Science
(1 million experiments)



Physics
(1 experiment)

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



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

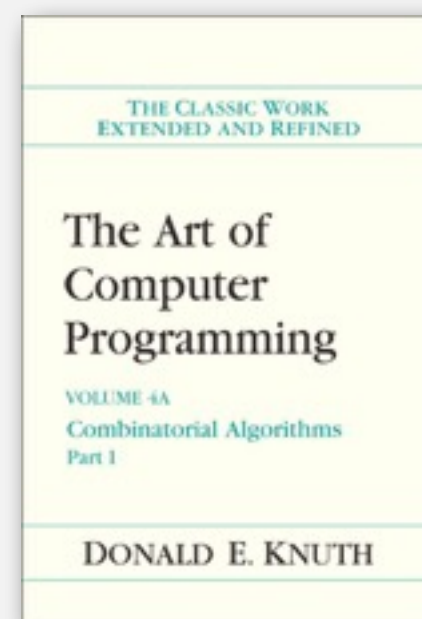
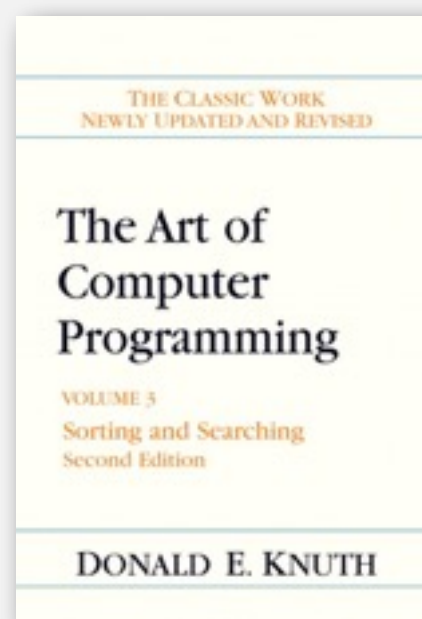
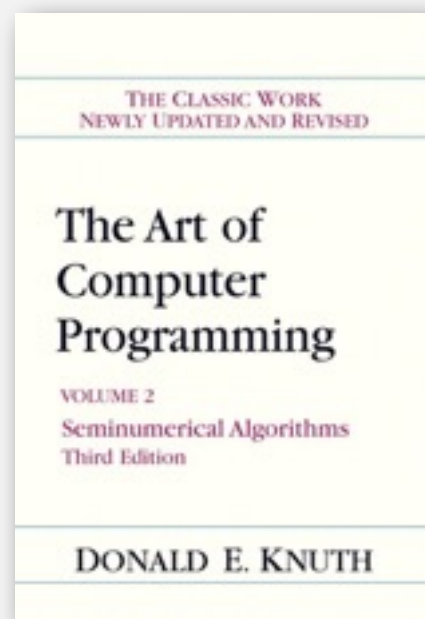
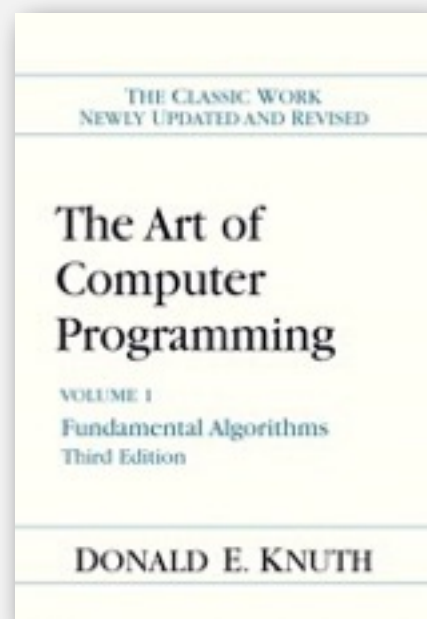
1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Mathematical models for running time

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

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Example: 1-SUM

Q. How many instructions as a function of input size N ?

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

 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 $2N$

† representative estimates (with some poetic license)

Example: 2-SUM

Q. How many instructions 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++;
```

Karl Friedrich Gauss, when he was 10 year's old and a schoolteacher made him add $1+2+3+\dots+100$

<https://www.youtube.com/watch?v=tpkzn2e5mtI>

$$\begin{aligned} 0 + 1 + 2 + \dots + (N-1) &= \frac{1}{2} N (N-1) \\ &= \binom{N}{2} \end{aligned}$$

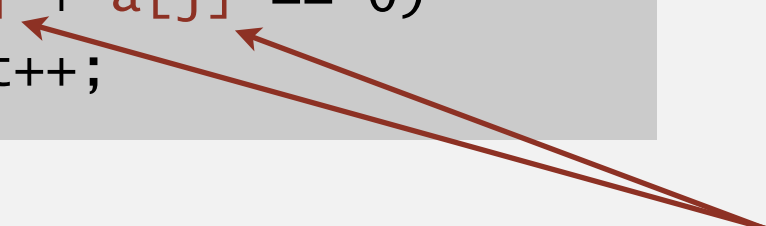
Pf. [Gauss]

$$\begin{array}{rcl} T(N) & = & 0 + 1 + \dots + (N-2) + (N-1) \\ + \quad T(N) & = & (N-1) + (N-2) + \dots + 1 + 0 \\ \hline 2 T(N) & = & (N-1) + (N-1) + \dots + (N-1) + (N-1) \\ \Rightarrow T(N) & = & N(N-1)/2 \end{array}$$

Example: 2-SUM

Q. How many instructions 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++;
```


$$0 + 1 + 2 + \dots + (N - 1) = \frac{1}{2} N (N - 1)$$
$$= \binom{N}{2}$$

operation	cost (ns)	frequency
variable declaration	2/5	$N + 2$
assignment statement	1/5	$N + 2$
less than compare	1/5	$\frac{1}{2} (N + 1) (N + 2)$
equal to compare	1/10	$\frac{1}{2} N (N - 1)$
array access	1/10	$N (N - 1)$
increment	1/10	$\frac{1}{2} N (N + 1)$ to N^2

$\frac{1}{4} N^2 + \frac{13}{20} N + \frac{13}{10} \text{ ns}$
to
 $\frac{3}{10} N^2 + \frac{3}{5} N + \frac{13}{10} \text{ ns}$
(tedious to count exactly)

Simplifying the calculations

*“ 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 given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and **we shall therefore only attempt to count the number of multiplications and recordings.** ” — Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(National Physical Laboratory, Teddington, Middlesex)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known ‘Gauss elimination process’, it is found that the errors are normally quite moderate: no exponential build-up need occur.



Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```
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++;
```

$$0 + 1 + 2 + \dots + (N - 1) = \frac{1}{2} N (N - 1) \\ = \binom{N}{2}$$

operation	cost (ns)	frequency
variable declaration	2/5	$N + 2$
assignment statement	1/5	$N + 2$
less than compare	1/5	$\frac{1}{2} (N + 1) (N + 2)$
equal to compare	1/10	$\frac{1}{2} N (N - 1)$
array access	1/10	$N (N - 1)$
increment	1/10	$\frac{1}{2} N (N + 1)$ to N^2

cost model = array accesses
(we assume compiler/JVM do not optimize any array accesses away!)

Simplification 2: tilde notation

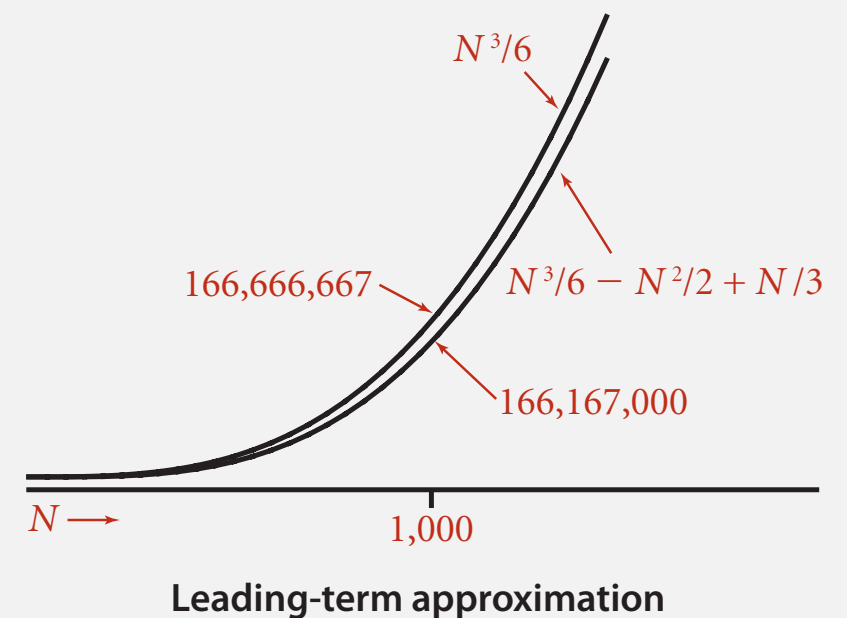
- Estimate running time (or memory) as a function of input size N .
- Ignore lower order terms.
 - when N is large, terms are negligible
 - when N is small, we don't care

Ex 1. $\frac{1}{6} N^3 + 20 N + 16 \sim \frac{1}{6} N^3$

Ex 2. $\frac{1}{6} N^3 + 100 N^{4/3} + 56 \sim \frac{1}{6} N^3$

Ex 3. $\frac{1}{6} N^3 - \underbrace{\frac{1}{2} N^2 + \frac{1}{3} N}_{\text{discard lower-order terms}} \sim \frac{1}{6} N^3$

(e.g., $N = 1000$: 166.67 million vs. 166.17 million)



Technical definition. $f(N) \sim g(N)$ means $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1$

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N .
- Ignore lower order terms.
 - when N is large, terms are negligible
 - when N is small, we don't care

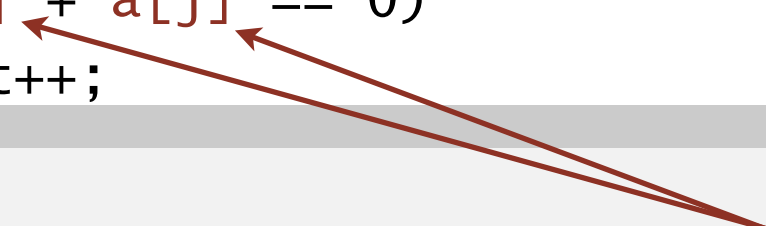
operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N (N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$\frac{1}{2} N (N + 1)$ to N^2	$\sim \frac{1}{2} N^2$ to $\sim N^2$

Example: 2-SUM

Q. Approximately how many array accesses 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"


$$\begin{aligned} 0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2} N (N - 1) \\ &= \binom{N}{2} \end{aligned}$$

A. $\sim N^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-SUM

Q. Approximately how many array accesses 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"

A. $\sim \frac{1}{2} N^3$ array accesses.

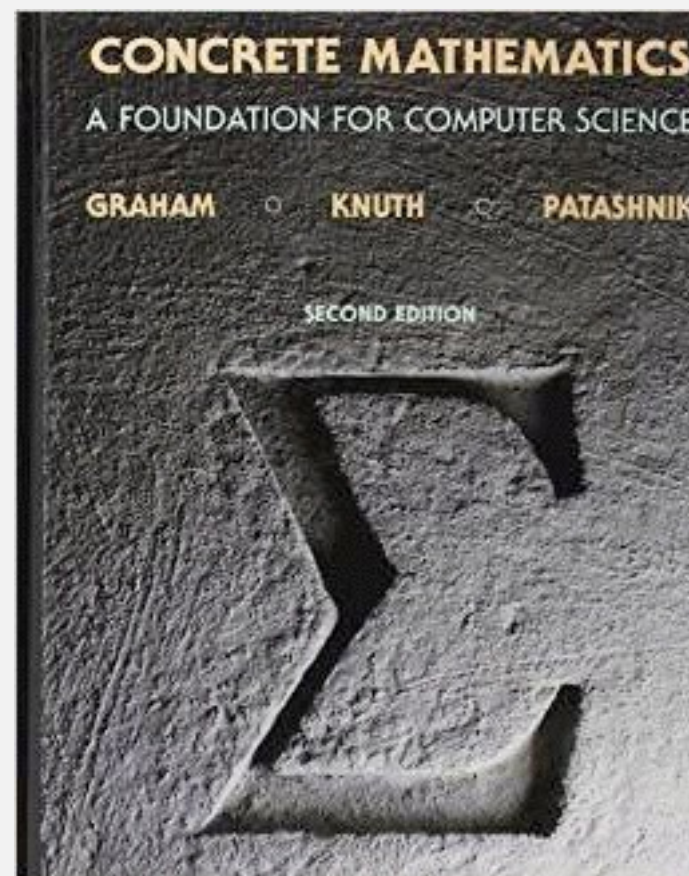
$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!}$$
$$\sim \frac{1}{6} N^3$$

Bottom line. Use cost model and tilde notation to simplify counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take a discrete mathematics course (CIS 160, 261).



Estimating a discrete sum

Q. How to estimate a discrete sum?

A2. Replace the sum with an integral, and use calculus!

Ex 1. $1 + 2 + \dots + N.$

$$\sum_{i=1}^N i \sim \int_{x=1}^N x \, dx \sim \frac{1}{2} N^2$$

Ex 2. $1 + 1/2 + 1/3 + \dots + 1/N.$

$$\sum_{i=1}^N \frac{1}{i} \sim \int_{x=1}^N \frac{1}{x} dx = \ln N$$

Ex 3. 3-sum triple loop.

$$\sum_{i=1}^N \sum_{j=i}^N \sum_{k=j}^N 1 \sim \int_{x=1}^N \int_{y=x}^N \int_{z=y}^N dz \, dy \, dx \sim \frac{1}{6} N^3$$

Ex 4. $1 + 1/2 + 1/4 + 1/8 + \dots$

$$\int_{x=0}^{\infty} \left(\frac{1}{2}\right)^x dx = \frac{1}{\ln 2} \approx 1.4427$$

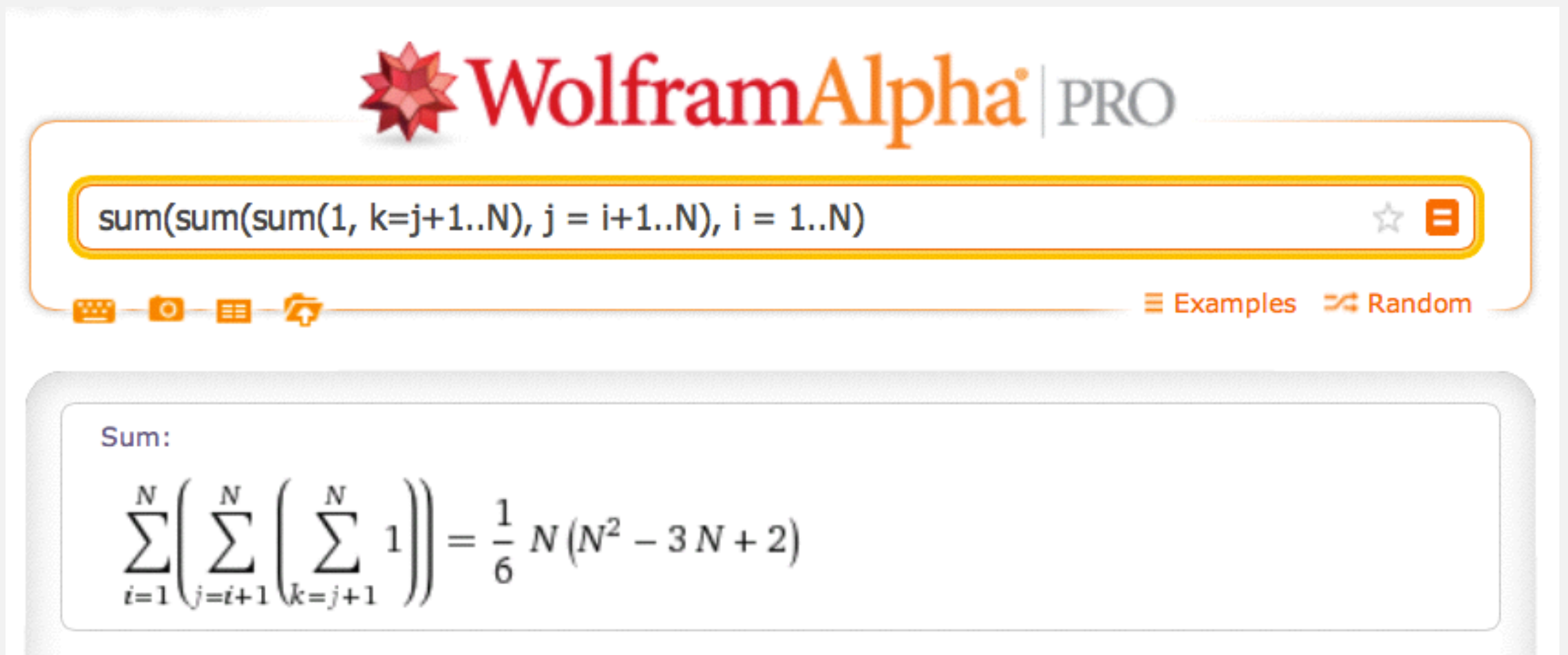
$$\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2$$

integral trick
doesn't always work!

Estimating a discrete sum

Q. How to estimate a discrete sum?

A3. Use Wolfram Alpha.



The screenshot shows the WolframAlpha PRO interface. At the top is the WolframAlpha logo with a red star icon and the word "PRO" in grey. Below the logo is a search bar with a yellow border containing the input: `sum(sum(sum(1, k=j+1..N), j = i+1..N), i = 1..N)`. To the right of the search bar are a star icon and an equals sign icon. Below the search bar is a row of icons: a keyboard, a camera, a list, and a share icon. To the right of these icons are the links "Examples" and "Random". Below the search bar is a large box containing the result. The word "Sum:" is in blue. Below it is the mathematical formula:
$$\sum_{i=1}^N \left(\sum_{j=i+1}^N \left(\sum_{k=j+1}^N 1 \right) \right) = \frac{1}{6} N (N^2 - 3 N + 2)$$

wolframalpha.com

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

A = array access

B = integer add

C = integer compare

D = increment

E = variable assignment

frequencies
(depend on algorithm, input)

Bottom line. We use **approximate** models in this course: $T(N) \sim c N^3$.



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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Common order-of-growth classifications


Definition. If $f(N) \sim c g(N)$ for some constant $c > 0$, then the **order of growth** of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the **running time** of this code is N^3 .

```
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++;
```

Typical usage. Mathematical analysis of running times.

 where leading coefficient
depends on machine, compiler, JVM, ...

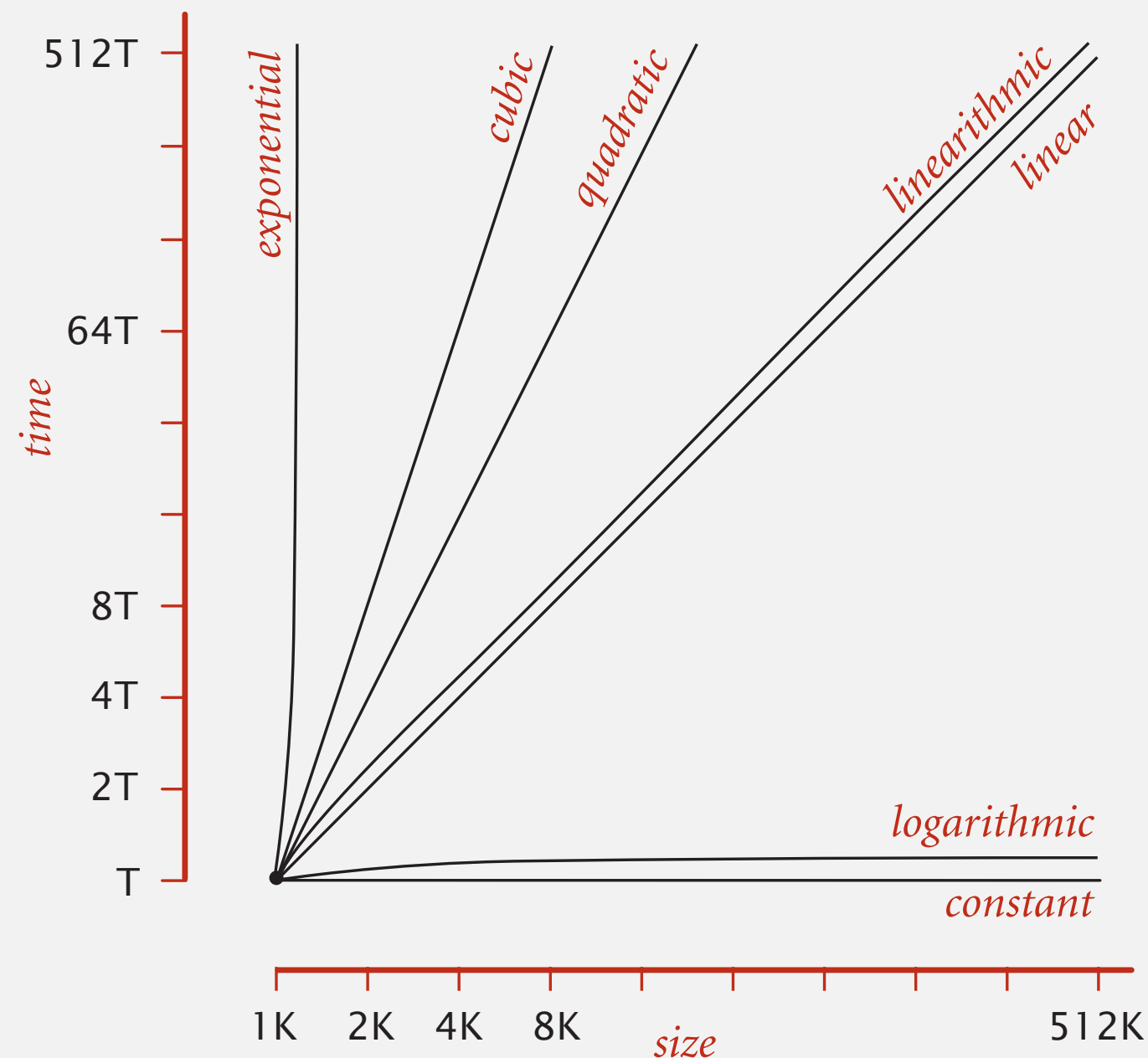
Common order-of-growth classifications

Good news. The set of functions

1 , $\log N$, N , $N \log N$, N^2 , N^3 , and 2^N

suffices to describe the order of growth of most common algorithms.

log-log plot



Typical orders of growth

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	logarithmic	<code>while (N > 1) { N = N/2; ... }</code>	divide in half	binary search	~ 1
N	linear	<code>for (int i = 0; i < N; i++) { ... }</code>	single loop	find the maximum	2
$N \log N$	linearithmic	<i>see mergesort lecture</i>	divide and conquer	mergesort	~ 2
N^2	quadratic	<code>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</code>	double loop	check all pairs	4
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	check all triples	8
2^N	exponential	<i>see combinatorial search lecture</i>	exhaustive search	check all subsets	2^N

Practical implications of order-of-growth

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	–	–
$\log N$	logarithmic	nearly independent of input size	–	–
N	linear	optimal for N inputs	a few minutes	100x
$N \log N$	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4–5x
2^N	exponential	useful only for tiny problems	forever	1x

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo							↑ mid							↑ hi

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑			↑			↑								
lo			mid			hi								

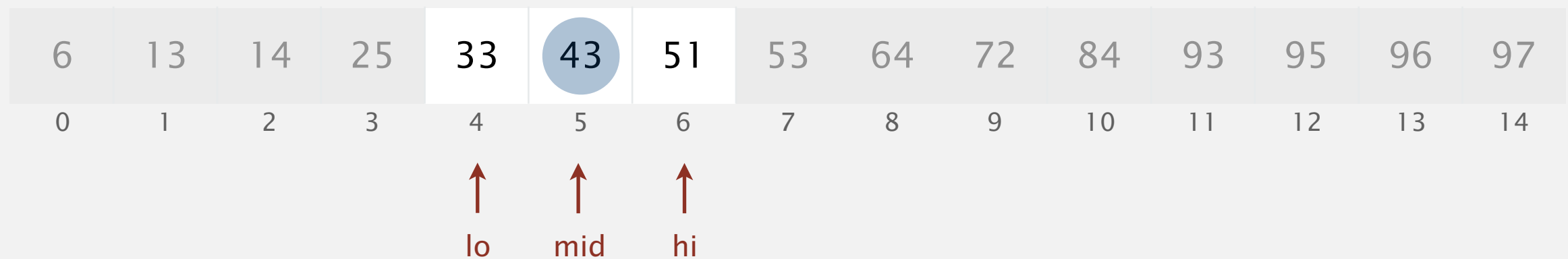
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

successful search for 33

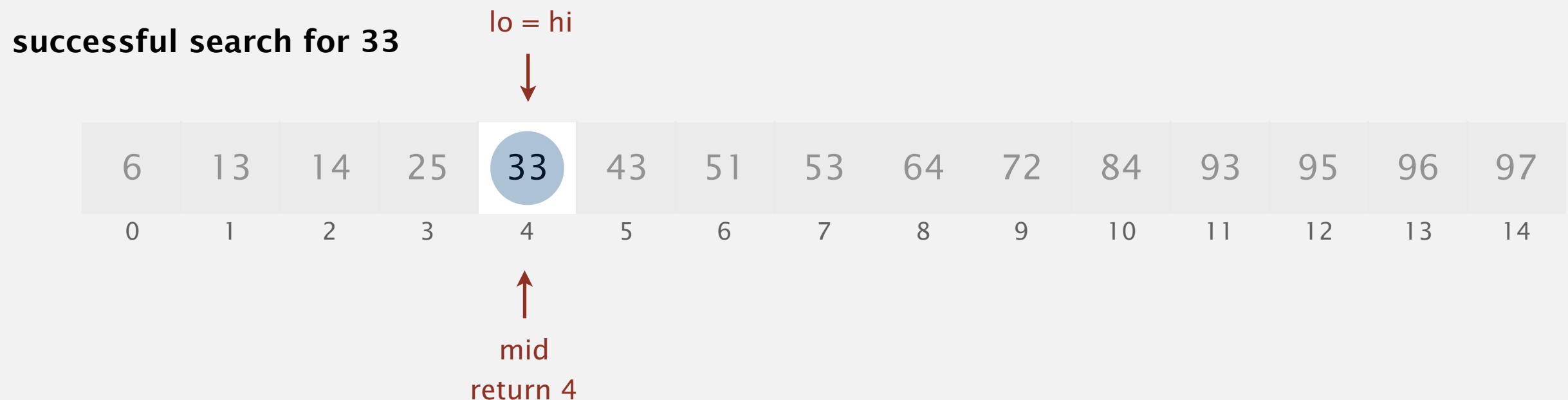


Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo							↑ mid							↑ hi

Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑			↑			↑								
lo			mid			hi								

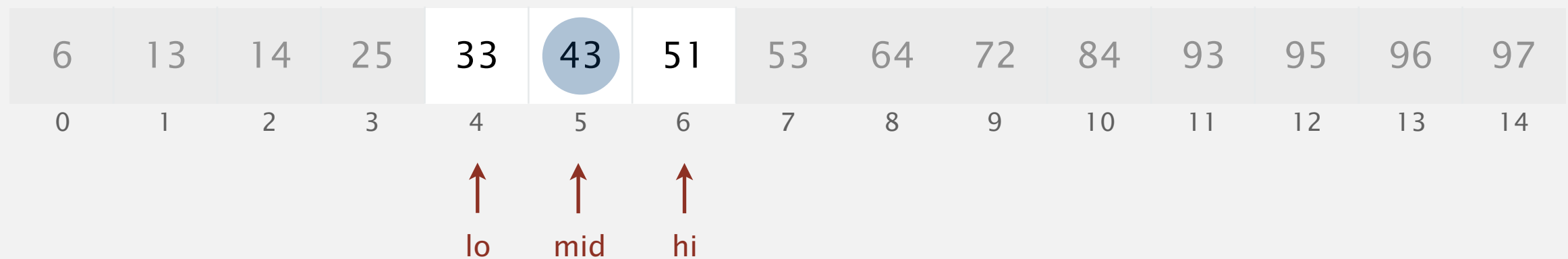
Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.

unsuccessful search for 34

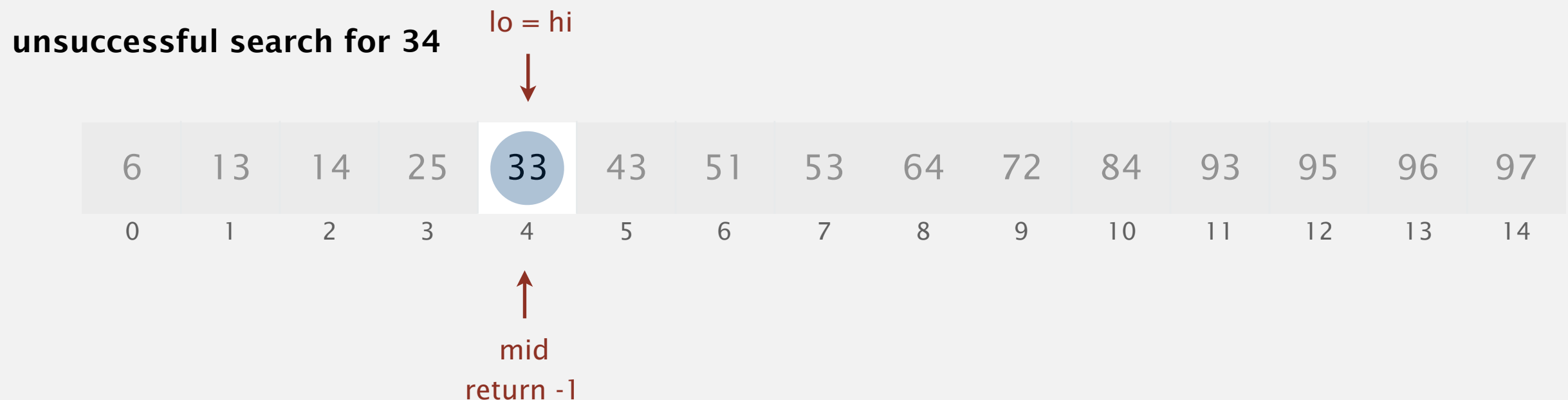


Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



Binary search: implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.



Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Posted by Joshua Bloch, Software Engineer

I remember vividly Jon Bentley's first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful *Programming Pearls* (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.




<http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>

Binary search: Java implementation

Invariant. If key appears in array $a[]$, then $a[lo] \leq key \leq a[hi]$.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length - 1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

why not $mid = (lo + hi) / 2$?



one "3-way compare"



Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ key compares to search in a sorted array of size N .

Def. $T(N)$ = # key compares to binary search a sorted subarray of size $\leq N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

\uparrow \uparrow
left or right half possible to implement with one
(floored division) 2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && \text{[given]} \\ &\leq T(N/4) + 1 + 1 && \text{[apply recurrence to first term]} \\ &\leq T(N/8) + 1 + 1 + 1 && \text{[apply recurrence to first term]} \\ &\vdots \\ &\leq T(N/N) + \underbrace{1 + 1 + \dots + 1}_{\lg N} && \text{[stop applying, } T(1) = 1 \text{]} \\ &= 1 + \lg N \end{aligned}$$

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth \Rightarrow faster in practice.



1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*



Types of analyses

Best case. Lower bound on cost.

- Determined by “easiest” input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

- Need a model for “random” input.
- Provides a way to predict performance.

this course

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. “Expected” cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

Theory of algorithms

Goals.

- Establish “difficulty” of a problem.
- Develop “optimal” algorithms.

Approach.

- Suppress details in analysis: analyze “to within a constant factor.”
- Eliminate variability in input model: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better.

Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ \vdots	classify algorithms
Big O	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ \vdots	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ \vdots	develop lower bounds

Theory of algorithms: example 1

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = “*Is there a 0 in the array?*”

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.

- Ex. **Improved** algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s–.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than “to within a constant factor” to predict performance.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10 N^2$	$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.

This course. Focus on approximate models: use Tilde-notation



1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ ***memory***

Basics

Bit. 0 or 1.

Byte. 8 bits.

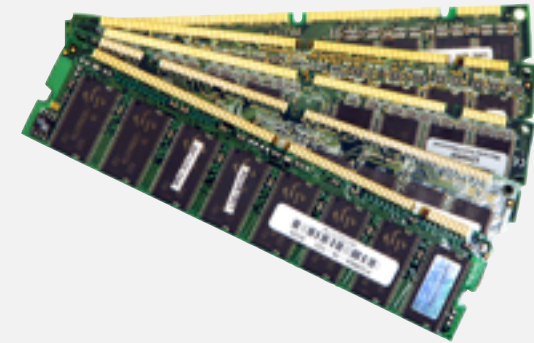
Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.

NIST



most computer scientists



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



some JVMs "compress" ordinary object 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
char[]	$2N + 24$
int[]	$4N + 24$
double[]	$8N + 24$

one-dimensional arrays

type	bytes
char[][]	$\sim 2MN$
int[][]	$\sim 4MN$
double[][]	$\sim 8MN$

two-dimensional arrays

Typical memory usage for objects in Java

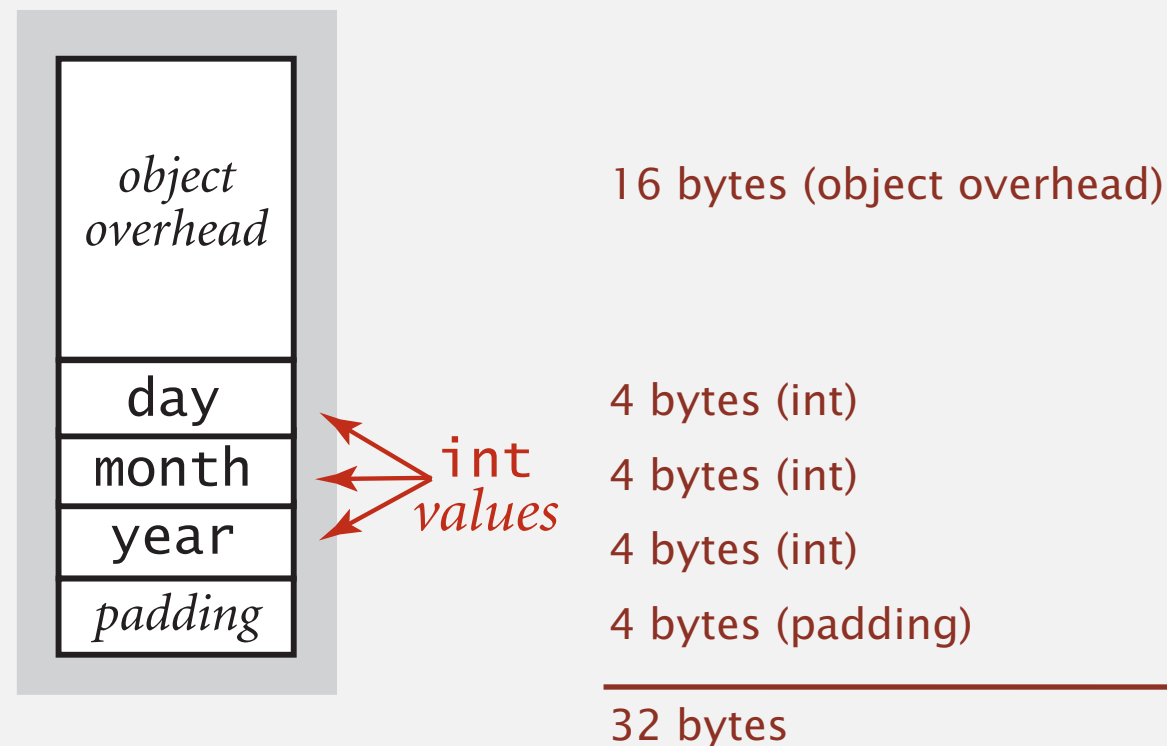
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 1. A Date object uses 32 bytes of memory.

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



Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

↑
+ 8 extra bytes per inner class object
(for reference to enclosing class)

Note. Depending on application, we may want to count memory for any referenced objects (recursively).

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 tilde notation to simplify analysis.
- Model enables us to **explain behavior**.

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

Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.

