SPRING 2023

INFORMATION TECHNOLOGY RESEARCH

YI HAN

DEPARTMENT OF INFORMATION MANAGEMENT
NATIONAL SUN YAT-SEN UNIVERSITY

Lecture slides are based on the supplemental materials of the textbook: https://algs4.cs.princeton.edu

Algorithms

FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

1.4 ANALYSIS OF ALGORITHMS

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

1.4 ANALYSIS OF ALGORITHMS

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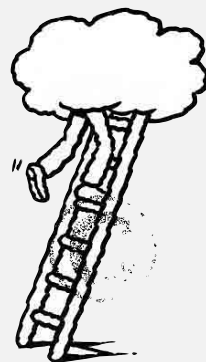- ▸ *theory of algorithms*
- ▸ *memory*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

Cast of characters

Programmer needs to develop a working solution.

Client wants to solve problem efficiently.

Theoretician wants to understand.

Student might play any or all of these roles someday.

Running time

" *As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?* "   — *Charles Babbage (1864)*
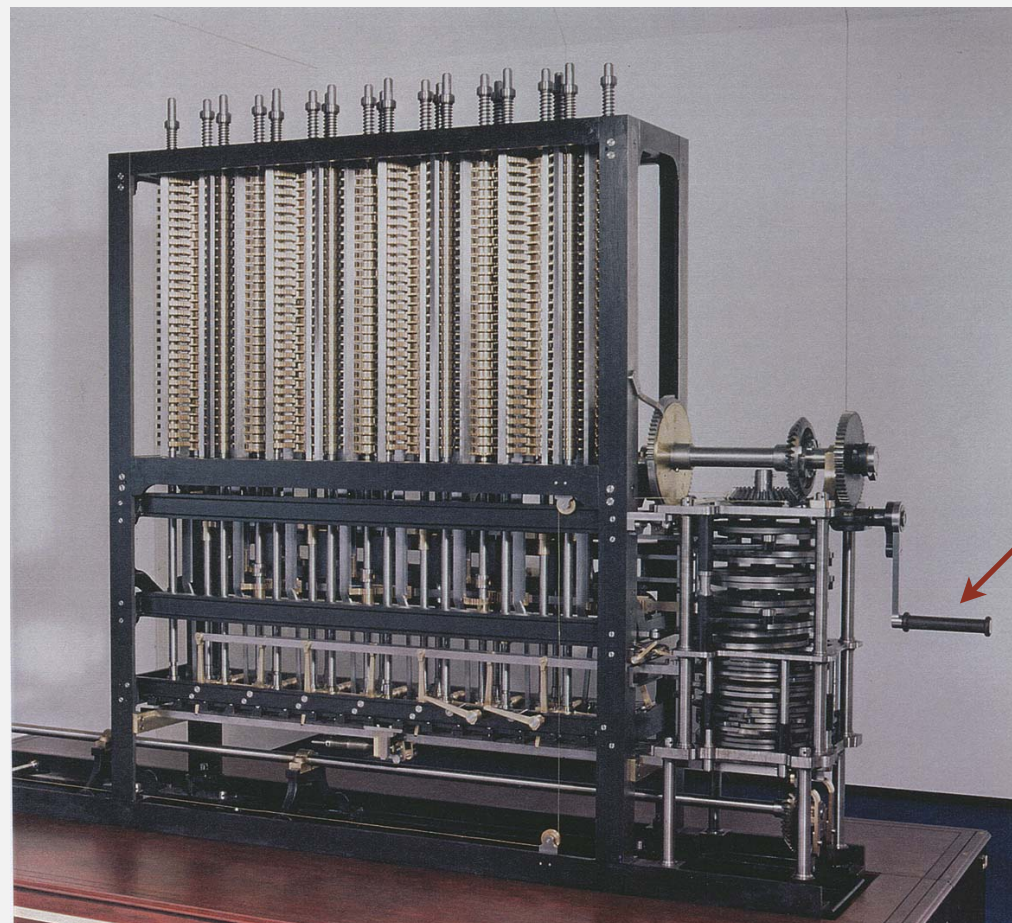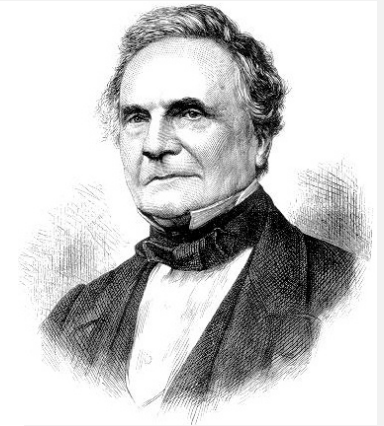
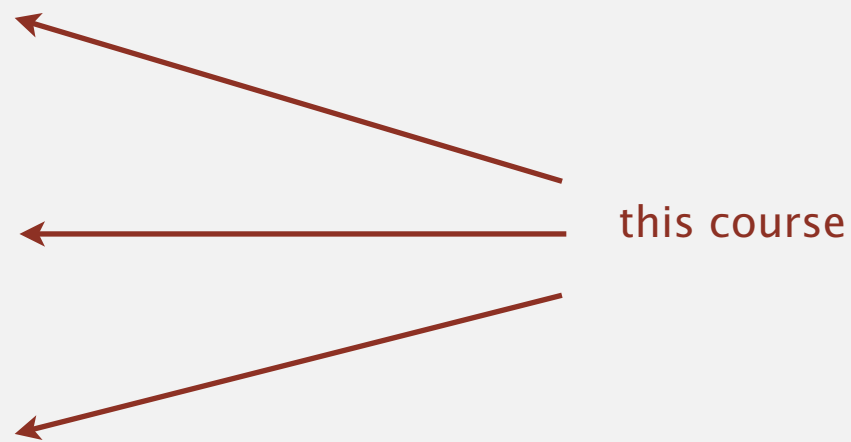how many times do you have to turn the crank?

Analytic Engine

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

this course

Provide guarantees.

Understand theoretical basis.

Primary practical reason: avoid performance bugs.

client gets poor performance because programmer did not understand performance characteristics

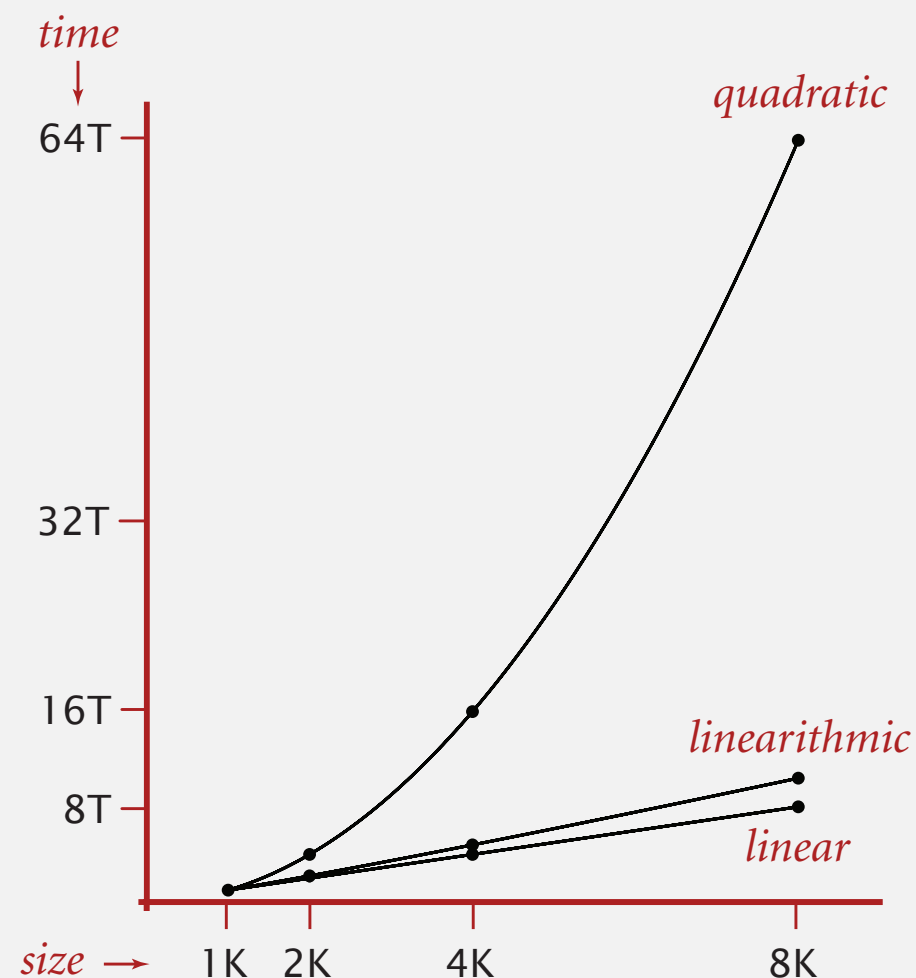# Some algorithmic successes

Discrete Fourier transform.

- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics, ...
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, enables new technology.

Friedrich Gauss
1805

time

64T —

quadratic

32T —

16T —

linearithmic

8T —

linear

size →   1K   2K   4K   8K

Some algorithmic successes

N-body simulation.

- Simulate gravitational interactions among N bodies.
- Brute force: N^2 steps.
- Barnes-Hut algorithm: $N \log N$ steps, enables new research.

Andrew Appel
PU '81

time

quadratic

64T

32T

16T

linearithmic

8T

linear

size → 1K 2K 4K 8K
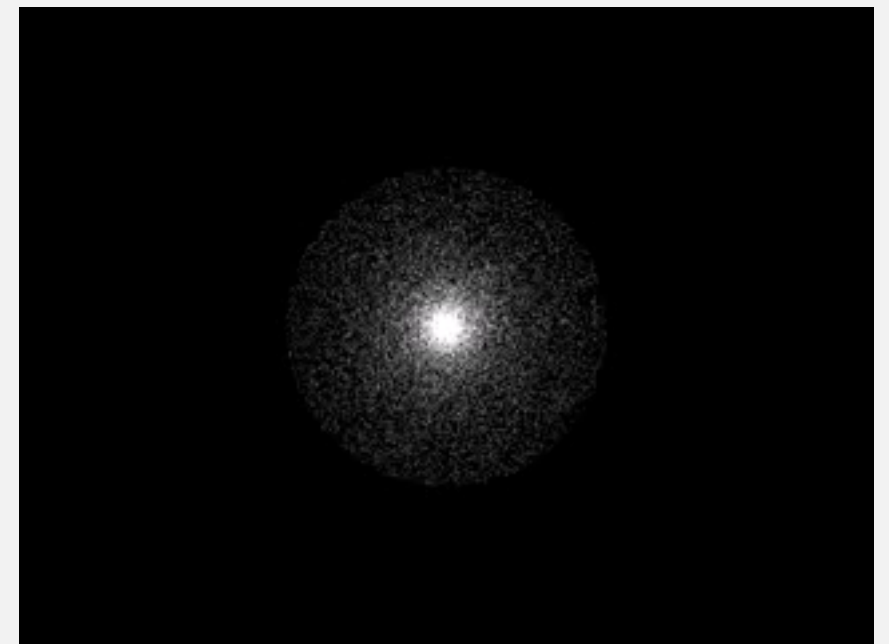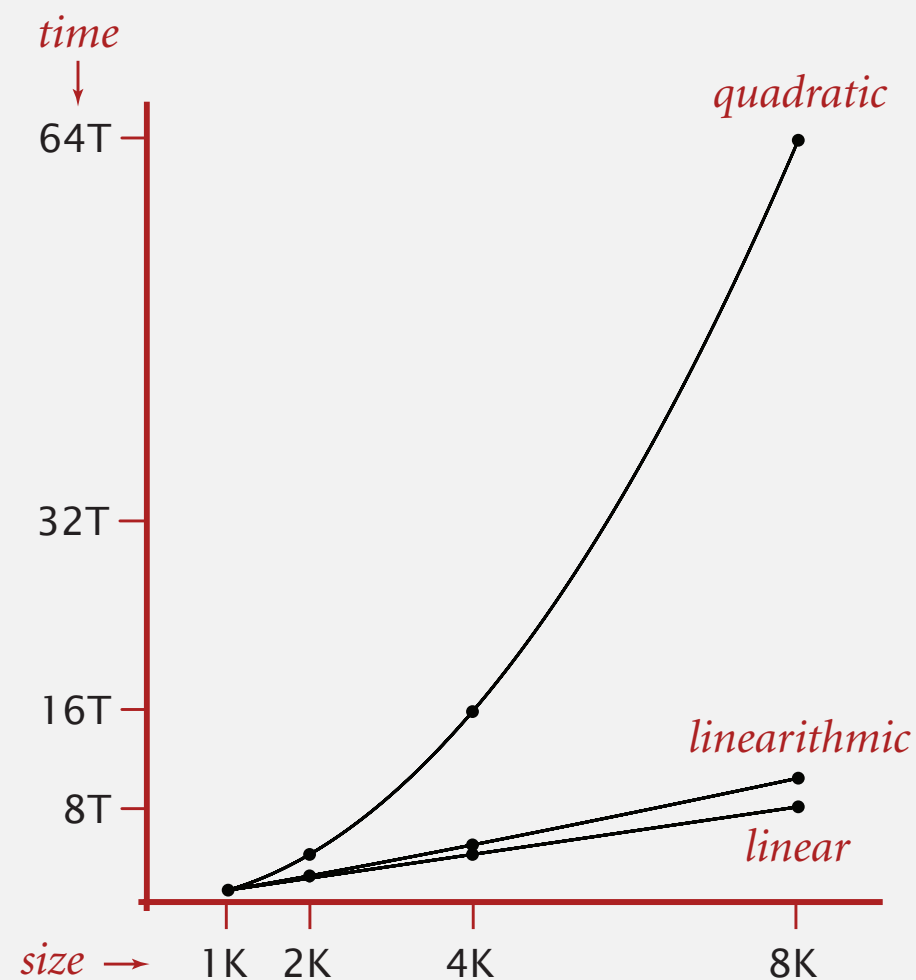
The challenge

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

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

Scientific method applied to 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.

Feature of the natural world. Computer itself.
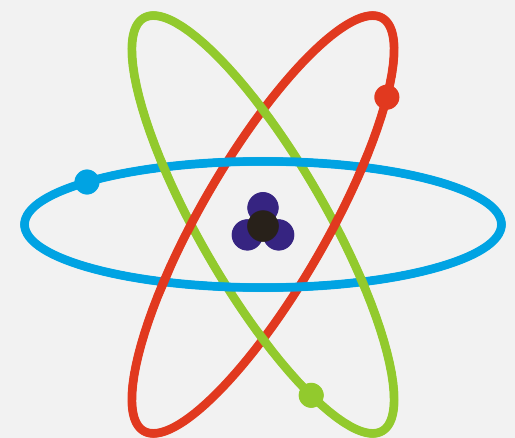
1.4 ANALYSIS OF ALGORITHMS

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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

```java
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));
    }
}
```
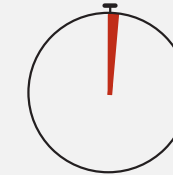
check each triple

Measuring the running time

Q. How to time a program?

A. Manual.

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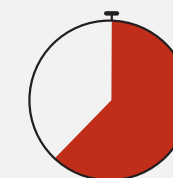

tick tick tick

70

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


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

528

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


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 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 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

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

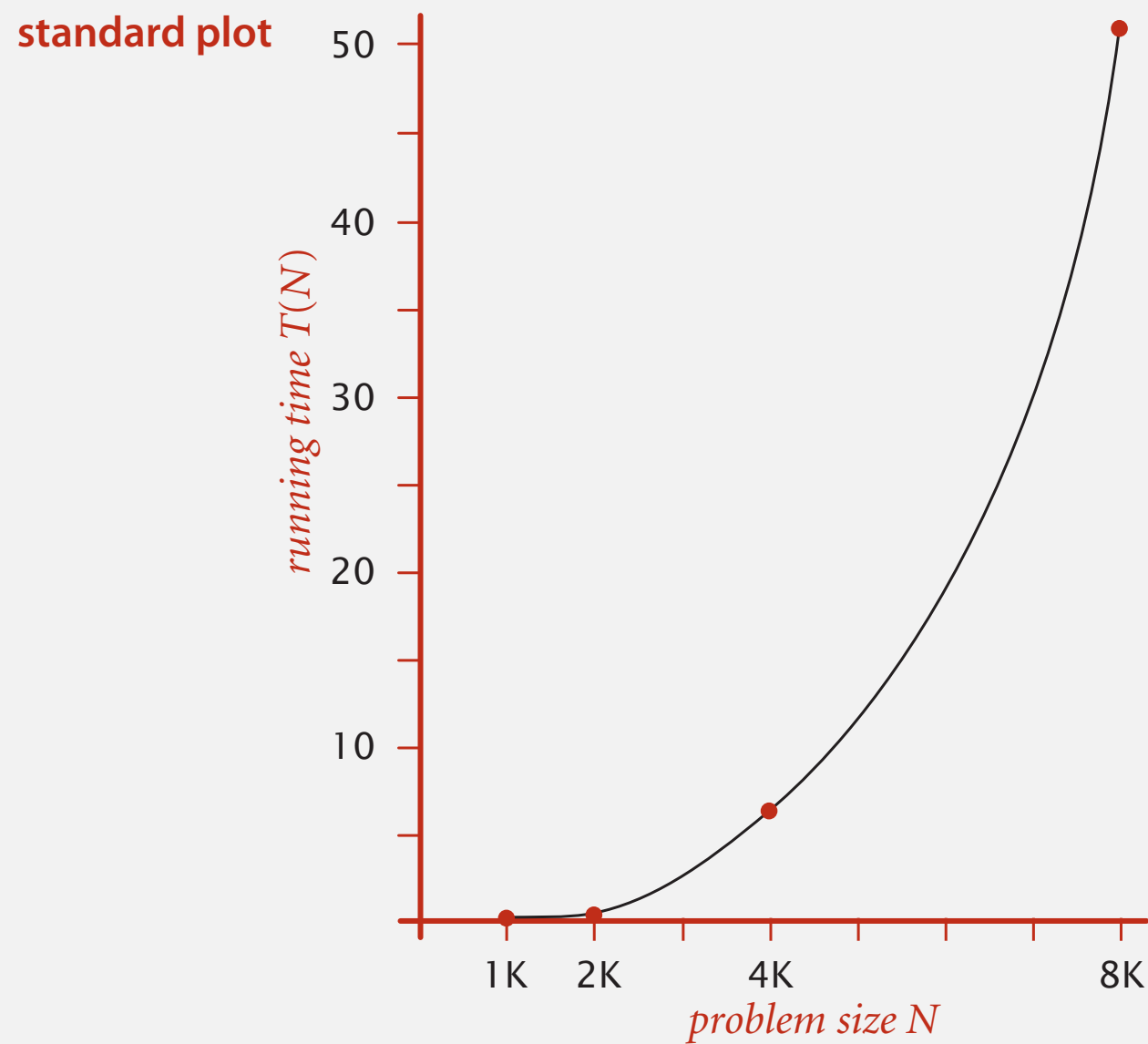Run the program for various input sizes and measure running time.

%

Empirical analysis

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

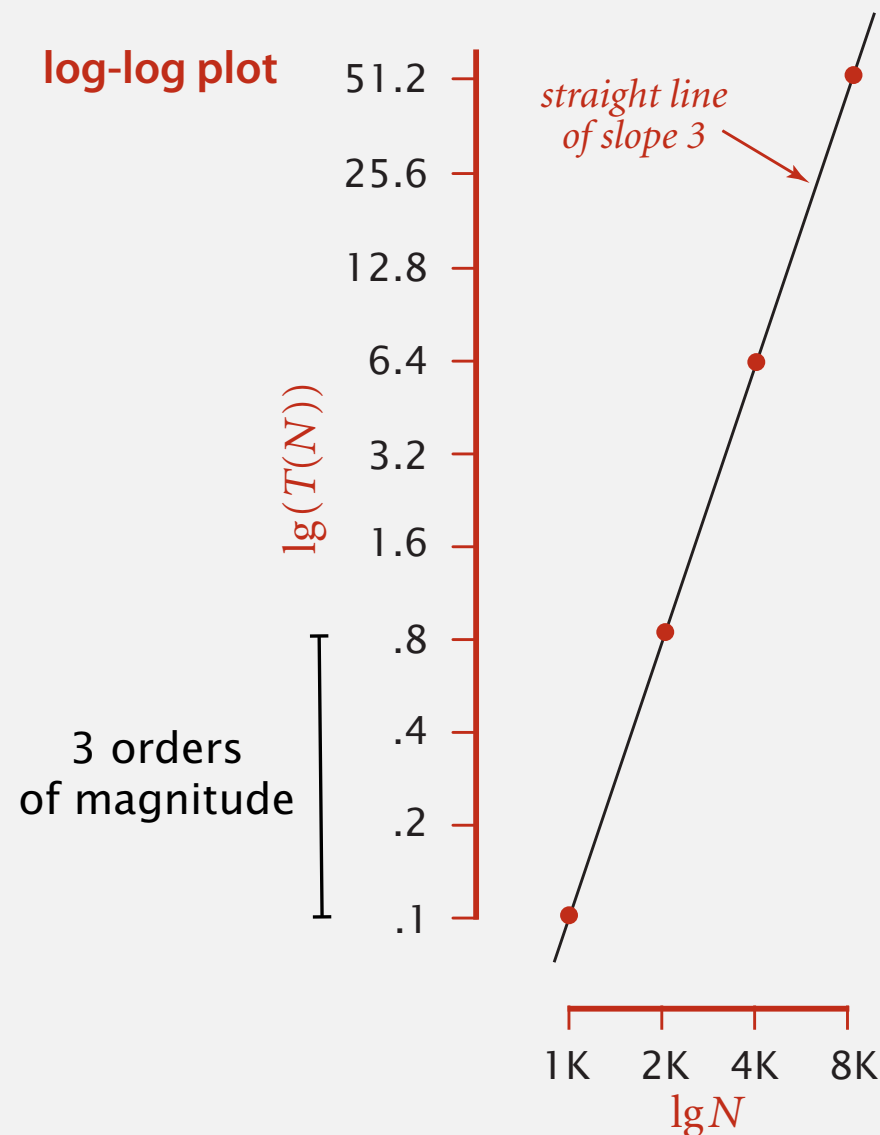| N	time (seconds) [†]
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

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.

log-log plot

straight line of slope 3

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

$b = 2.999$

$c = -33.2103$

$T(N) = a N^b$, *where* $a = 2^c$

3 orders of magnitude

1K 2K 4K 8K

$\lg N$

$\lg(T(N))$

power law

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

slope

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

Prediction and validation

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

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

lg(T(2N)/T(N))
lg (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

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

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

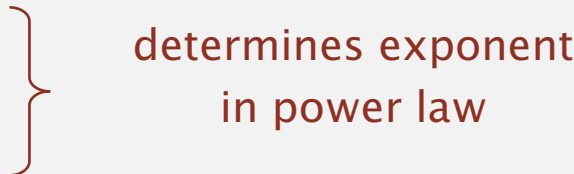$\Rightarrow \ a \ = \ 0.998 \times 10^{-10}$

Hypothesis. Running time is about $0.998 \times 10^{-10} \times N^3$ seconds.
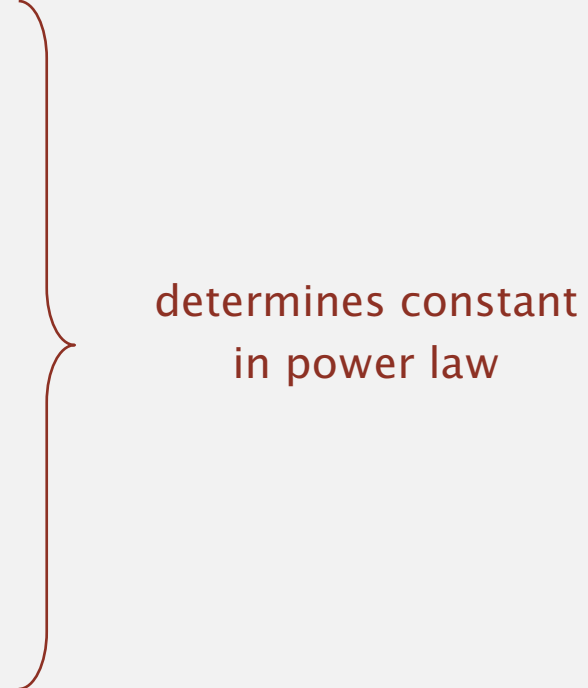
almost identical hypothesis
to one obtained via linear regression

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent
in power law

System dependent effects.

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

determines constant
in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

e.g., can run huge number of experiments

1.4 ANALYSIS OF ALGORITHMS

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

Mathematical models for running time

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

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.
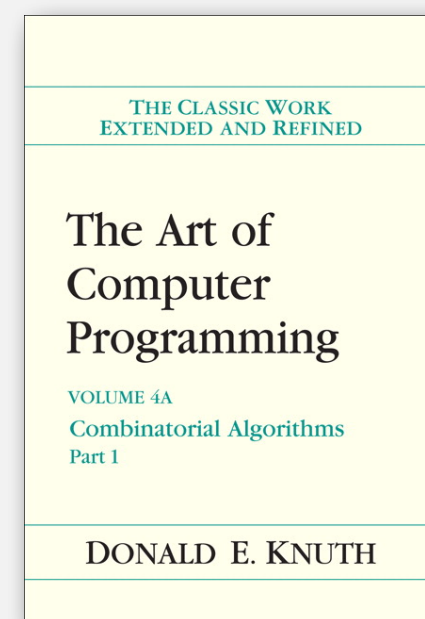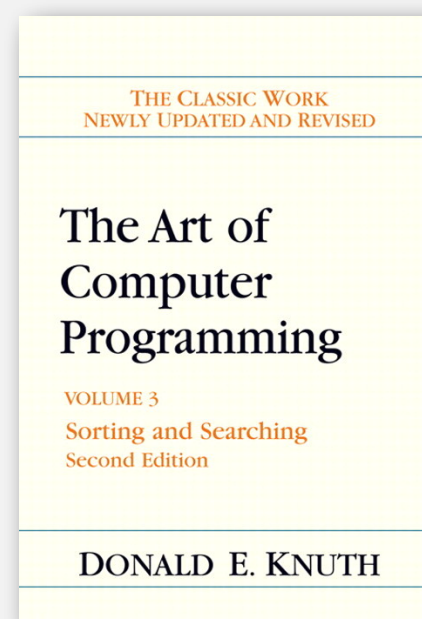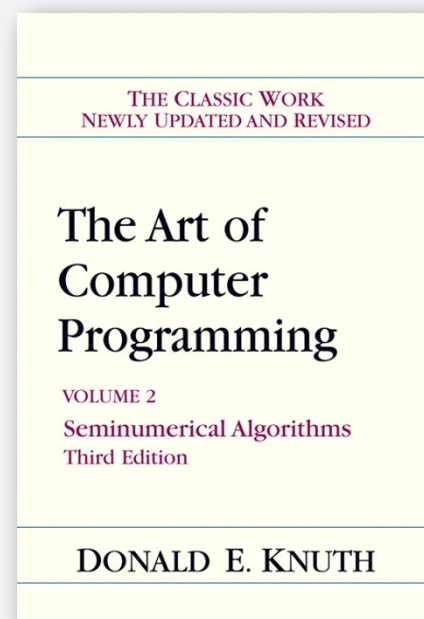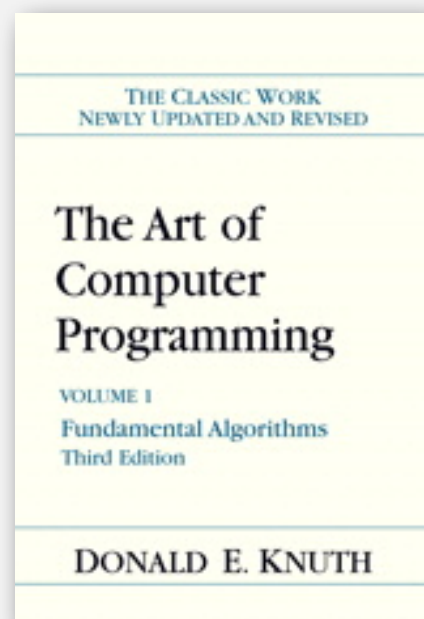
Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

Challenge. How to estimate constants.

operation	example	nanoseconds [†]
integer add	a + b	2.1
integer multiply	a * b	2.4
integer divide	a / b	5.4
floating-point add	a + b	4.6
floating-point multiply	a * b	4.2
floating-point divide	a / b	13.5
sine	Math.sin(theta)	91.3
arctangent	Math.atan2(y, x)	129
…	…	…

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

Observation. Most primitive operations take constant time.

operation	example	nanoseconds [†]
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6 N$
2D array allocation	`new int[N][N]`	$c_7 N^2$

Caveat. Non-primitive operations often take more than constant time.

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++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	
equal to compare	?
array access	
increment	

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++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	
array access	?
increment	

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++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	?
increment	

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	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	?

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	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2\,N$

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++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;=\; \frac{1}{2}N(N-1)$$

Pf. [n even]
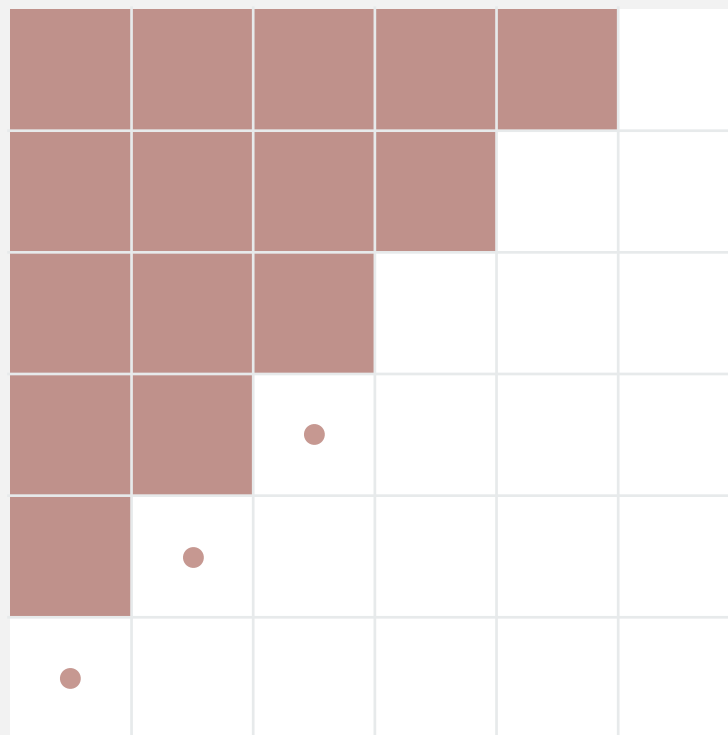
$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2}N^2 - \frac{1}{2}N$$

half of   half of
square   diagonal

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++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;=\; \frac{1}{2} N (N-1)$$

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

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
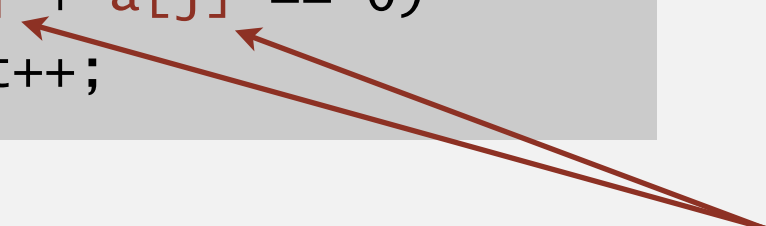
A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite 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++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;\; = \;\; \frac{1}{2}\,N\,(N-1)$$
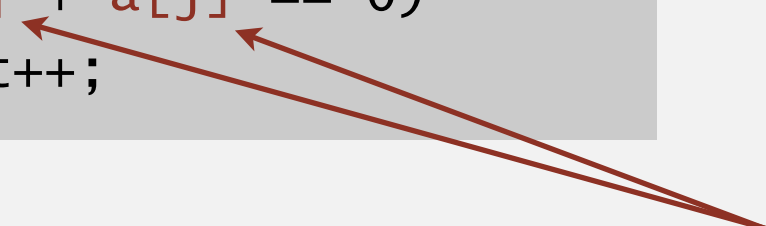
operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}\,(N+1)\,(N+2)$
equal to compare	$\frac{1}{2}\,N\,(N-1)$
array access	$N\,(N-1)$
increment	$\frac{1}{2}\,N\,(N-1)$ to $N\,(N-1)$

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. $\frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N$ $\sim \frac{1}{6} N^3$

discard lower-order terms
(e.g., N = 1000: 166.67 million vs. 166.17 million)

$N^3/6$

$166,666,667$ $N^3/6 - N^2/2 + N/3$

$166,167,000$

$N \longrightarrow$ $1,000$

Leading-term approximation

Technical definition. $f(N) \sim g(N)$ means $\displaystyle \lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$
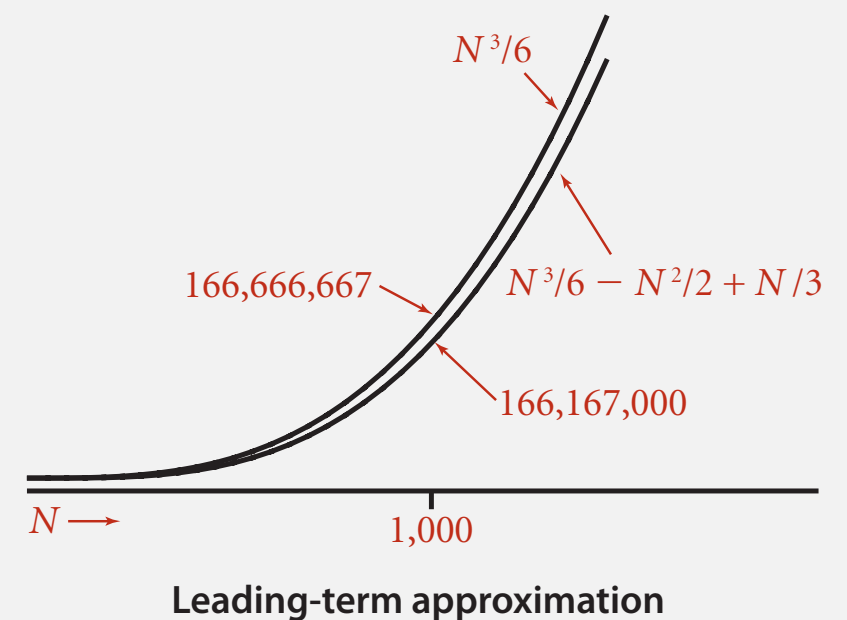
Simplification 2: tilde notation

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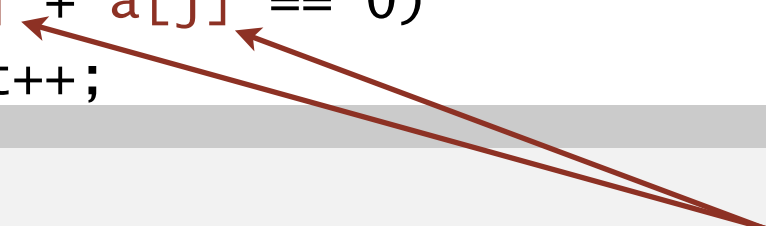
operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}(N+1)(N+2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2}N(N-1)$	$\sim \frac{1}{2} N^2$
array access	$N(N-1)$	$\sim N^2$
increment	$\frac{1}{2}N(N-1)$ to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

$$0 + 1 + 2 + \ldots + (N - 1) \;\; = \;\; \frac{1}{2} N (N - 1)$$

A. ~ 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"

$$= \frac{N(N-1)(N-2)}{3!}$$

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

$$\sim \frac{1}{6}N^3$$

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

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: $T(N) \sim c\, N^3$.

1.4 ANALYSIS OF ALGORITHMS

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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. With running times.

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

Common order-of-growth classifications

Good news. The set of functions

$$1, \quad \log N, \quad N, \quad N \log N, \quad N^2, \quad N^3, \text{ and } 2^N$$

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

log-log plot

Typical orders of growth
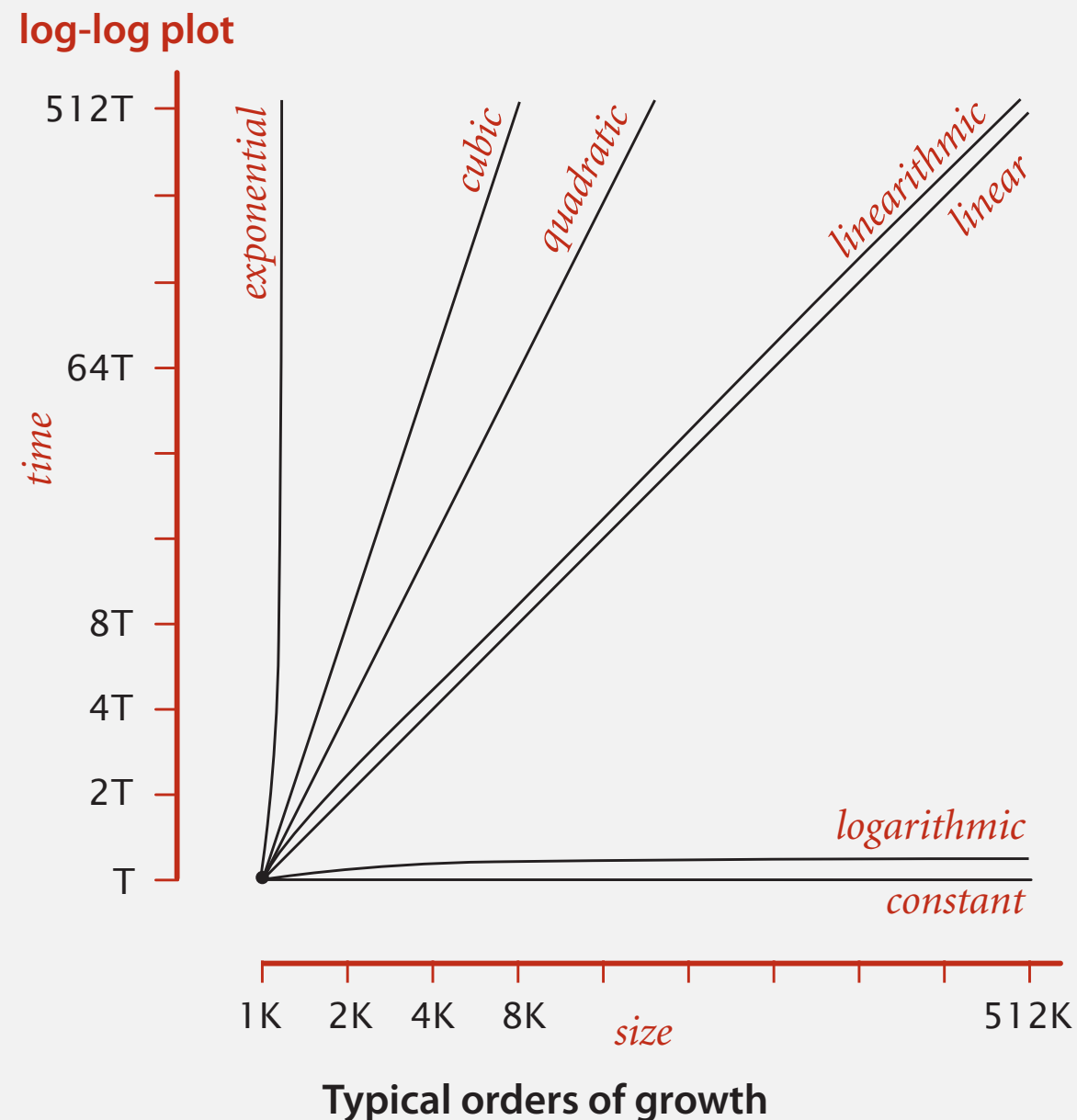
Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1)` `{    N = N / 2;    ...    }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++)` `{   ...        }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2

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

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

2^N	**exponential**	...	exhaustive search	check all subsets	$T(N)$

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

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

lo = hi

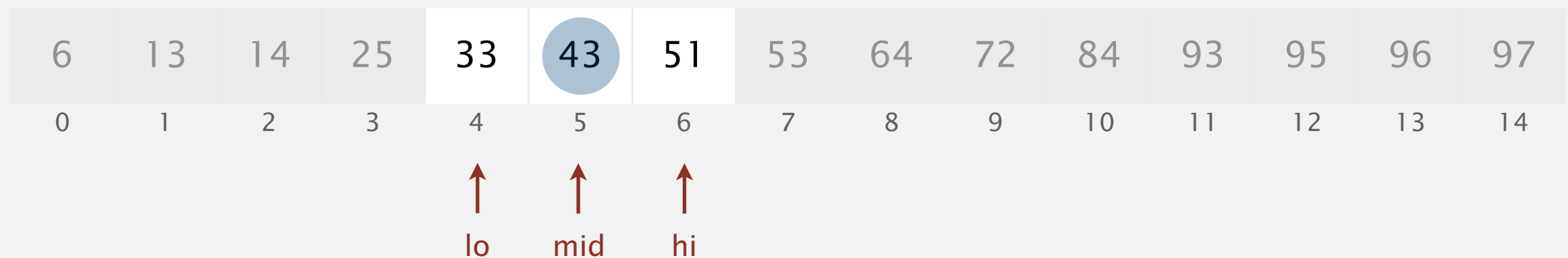| 6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

mid
return 4

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

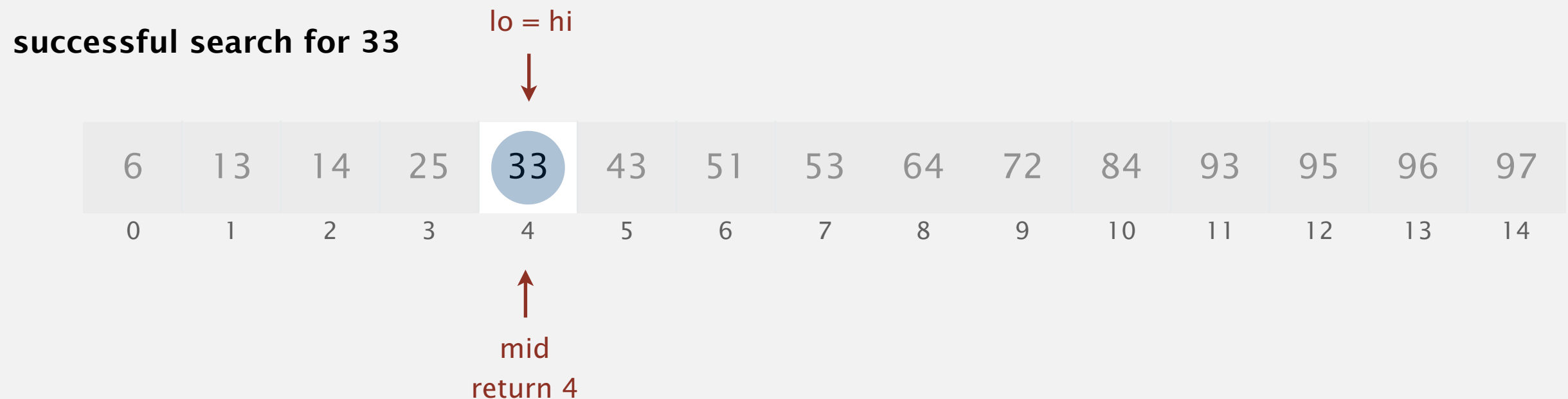↑ lo   ↑ mid   ↑ hi

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

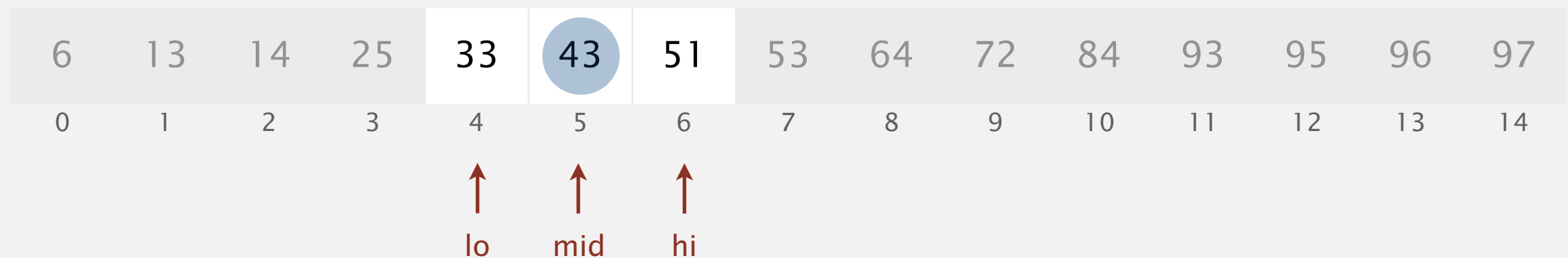| 6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑   ↑   ↑

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
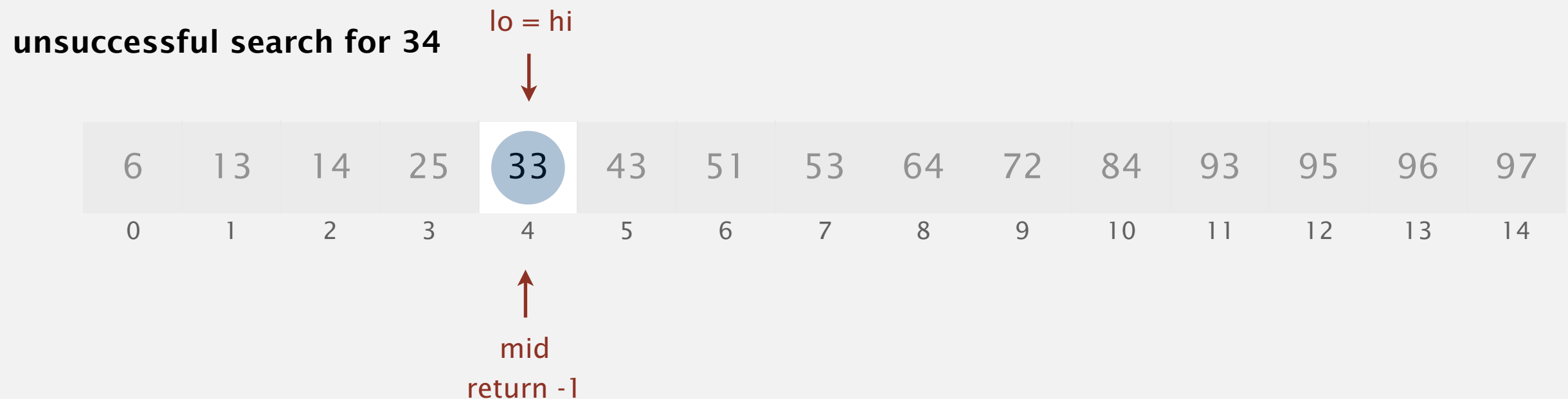
lo = hi

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

mid
return -1

Binary search: Java 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.

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


      What to write here? 3 mins.


   }
   return -1;
}
```

Invariant. If `key` appears in the array `a[]`, then `a[lo]` ≤ key ≤ `a[hi]`.

Binary search: Java 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.

```java
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;     ← one "3-way compare"
        else return mid;
    }
    return -1;
}
```

Invariant. If `key` appears in the array `a[]`, then `a[lo]` ≤ key ≤ `a[hi]`.

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 $T(1) = 1$.

↑ left or right half (floored division)

↑ possible to implement with one 2-way compare (instead of 3-way)

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

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

An N² log N algorithm for 3-Sum

Algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers `a[i]` and `a[j]`, binary search for `-(a[i] + a[j])`.

Analysis. Order of growth is $N^2 \log N$.

- Step 1: N^2 with insertion sort.
- Step 2: $N^2 \log N$ with binary search.

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10   0 5 10 30 40

binary search

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-20, -10)	30
⋮	⋮
(-10, 0)	10
⋮	⋮
(10, 30)	-40
(10, 40)	-50
(30, 40)	-70

only count if
a[i] < a[j] < a[k]
to avoid
double counting

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

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

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.

this course

Average case. Expected cost for random input.
- Need a model for "random" input.
- Provides a way to predict performance.

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:	$\sim \lg N$

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	$\Theta(N^2)$	$\tfrac{1}{2}\,N^{2}$ $10\,N^{2}$ $5\,N^{2}+22\,N\log N+3\mathrm{N}$ ⋮	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	$\Omega(N^2)$	$\tfrac{1}{2}\,N^{2}$ N^{5} $N^{3}+22\,N\log N+3\,N$ ⋮	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?

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?

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	$\Theta(N^2)$	$\tfrac{1}{2}\,N^2$ $10\,N^2$ $5\,N^2 + 22\,N \log N + 3N$	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	$\Omega(N^2)$	$\tfrac{1}{2}\,N^2$ 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

Basics

Bit. 0 or 1.
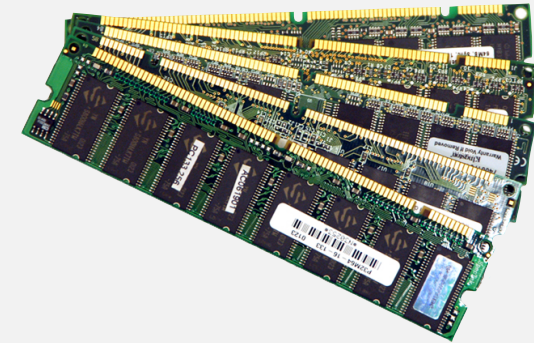
Byte. 8 bits.

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.

- Can address more memory.
- Pointers use more space.

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

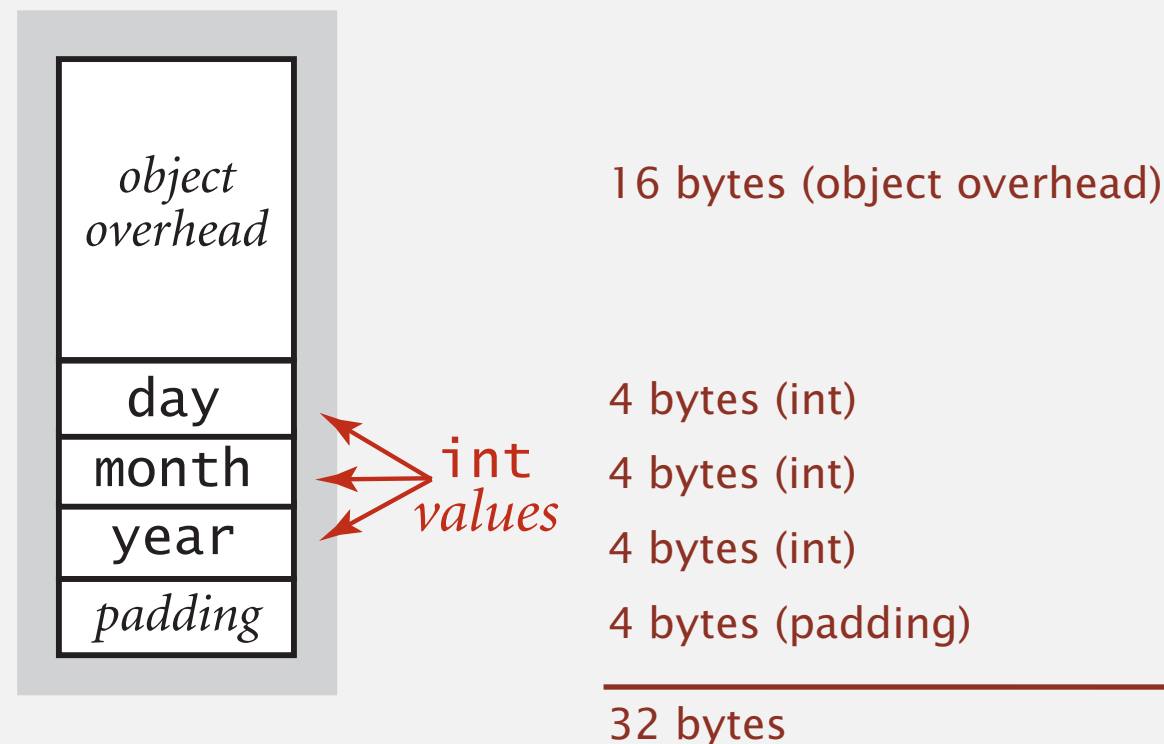# Typical memory usage for objects in Java

Object overhead. 16 bytes.

Reference. 8 bytes. To class, included in overhead.

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;
...
}
```

object overhead	16 bytes (object overhead)
day	4 bytes (int)
month → int *values*	4 bytes (int)
year	4 bytes (int)
padding	4 bytes (padding)
	32 bytes

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.
- Object: 16 bytes + memory for each instance variable.
- Array: 24 bytes (incl. length) + memory for each array entry.
- Padding: round up to multiple of 8 bytes.

Example

Q. How much memory does `WeightedQuickUnionUF` use as a function of N? Use tilde notation to simplify your answer.

```
public class WeightedQuickUnionUF
{
    private int[] id;
    private int[] sz;
    private int count;

    public WeightedQuickUnionUF(int N)
    {
        id = new int[N];
        sz = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
        for (int i = 0; i < N; i++) sz[i] = 1;
    }
    ...
}
```

16 bytes
(object overhead)

8 + (4N + 24) bytes each
(reference + `int[]` array)

4 bytes (`int`)

4 bytes (padding)

8N + 88 bytes

A. $8N + 88 \sim 8N$ bytes.

Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and 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.
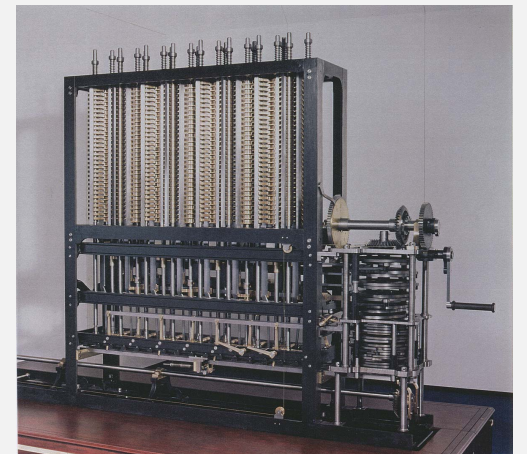
Scientific method.

- 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.