Analysis of Algorithms

CS 121: Data Structures

START RECORDING

Outline

- Attendance quiz
- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Theory of algorithms
- Memory

Attendance Quiz

Attendance Quiz: Creating ADTs

- Scan the QR code, or find today's attendance quiz under the "Quizzes" tab on Canvas
- Password: to be announced in class
- Implement a class, Luminosity, which includes the listed methods

Attendance Quiz: Creating ADTs

- Write your name
- Implement a class, Luminosity, which includes the following methods

```
public class Luminosity

Luminosity(int l)       // Accepts luminosity values from 0 to 255
int getLuminosity()     // Get l, the luminosity value
Luminosity brighter()   // Brighter version of this Luminosity
Luminosity darker()     // Darker version of this Luminosity
String toString()       // String representation
boolean equals(Luminosity m) // Is this luminosity the same as m's ?
```

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

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

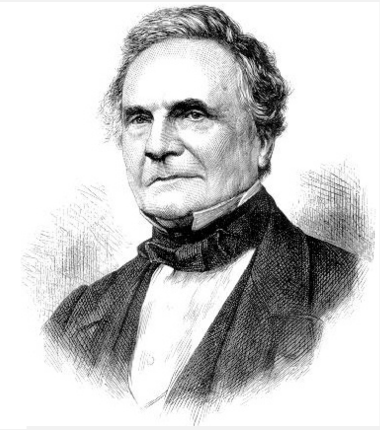
http://algs4.cs.princeton.edu

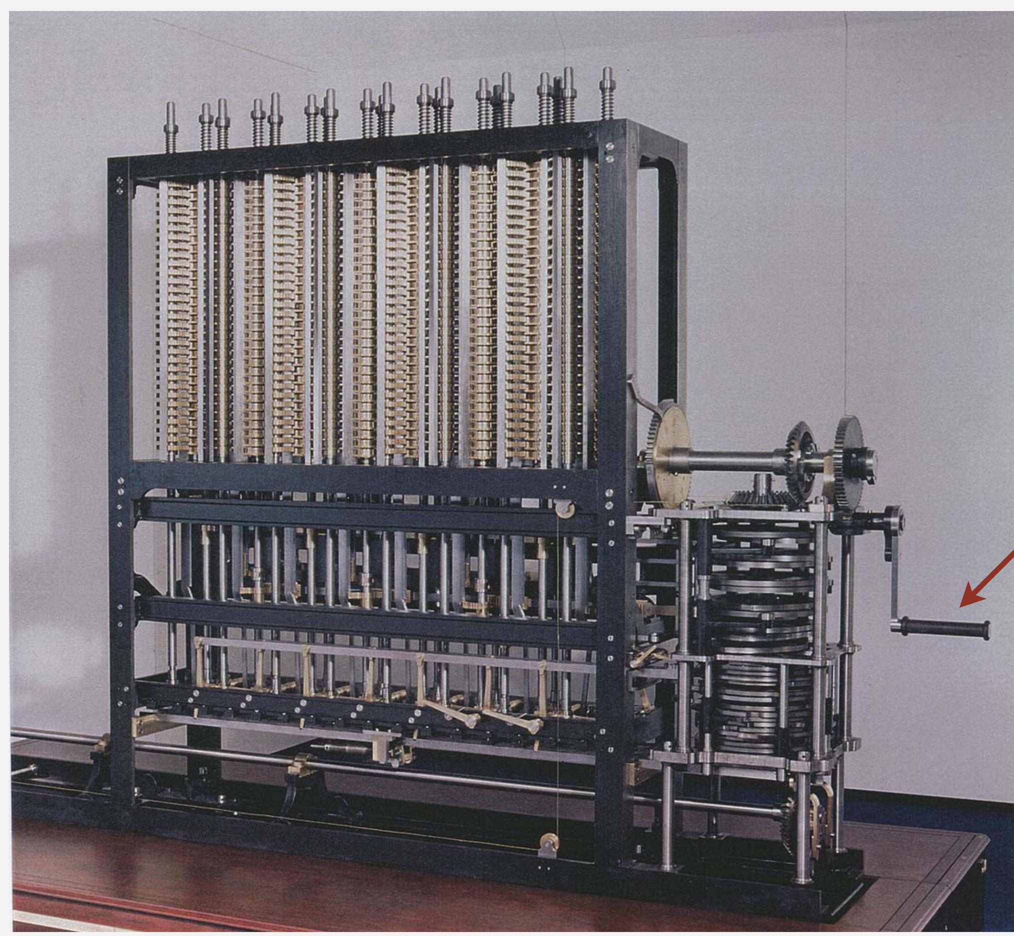
1.4 Analysis of Algorithms

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

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?

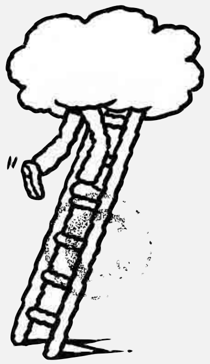
Analytic Engine

Cast of characters

Programmer needs to develop a working solution.

Student might play any or all of these roles someday.

Client wants to solve problem efficiently.



Theoretician wants to understand.

Reasons to analyze algorithms

Predict performance.

Compare algorithms. this course

Provide guarantees.

Understand theoretical basis.

Primary practical reason: avoid performance bugs.

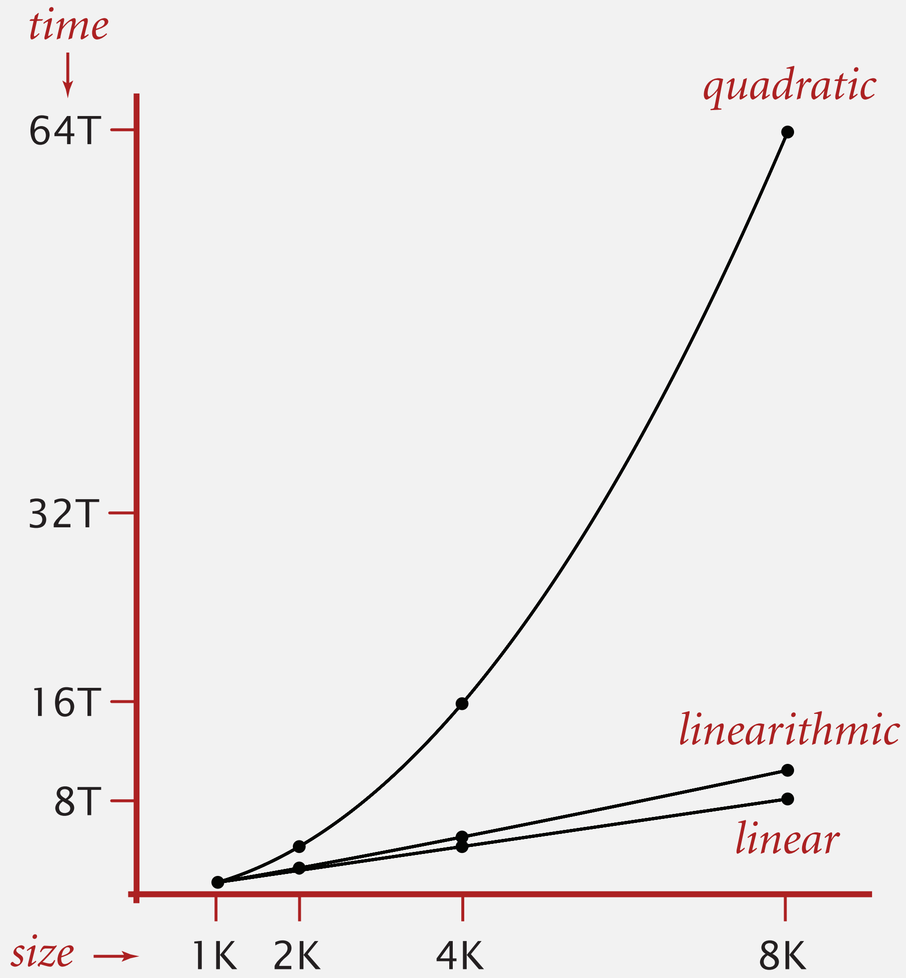
client gets poor performance because programmer did not understand performance characteristics

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

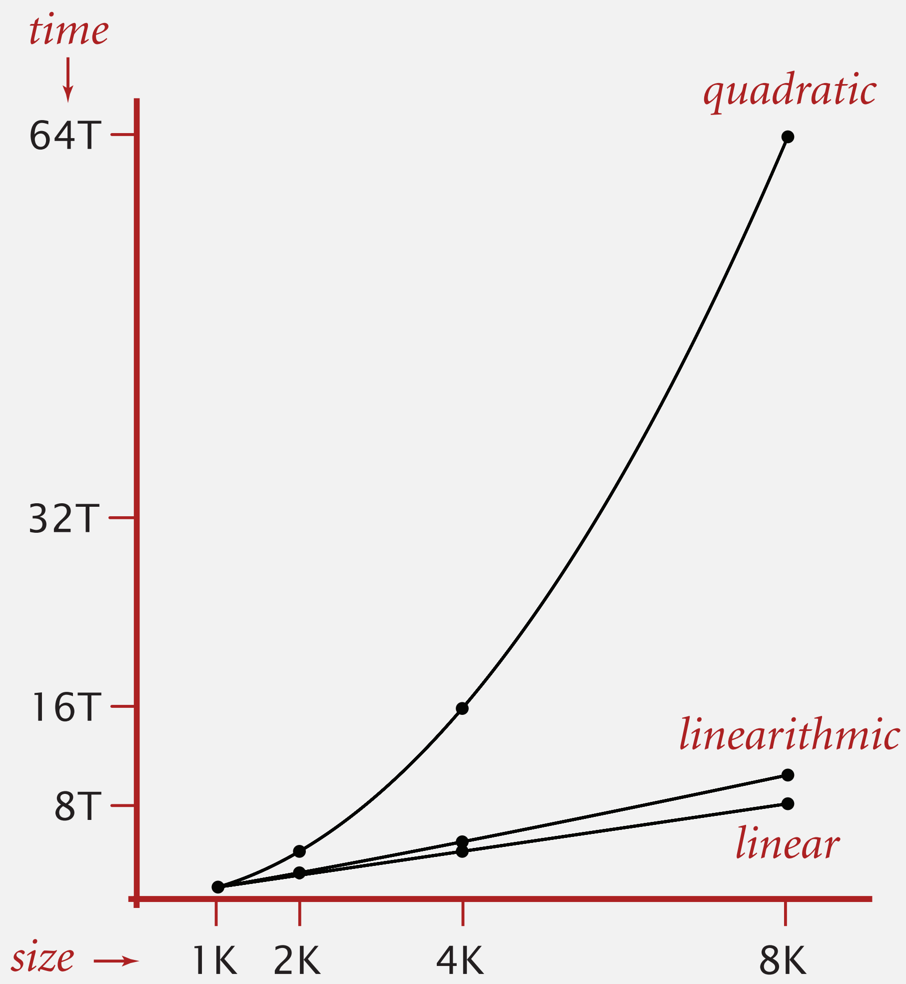


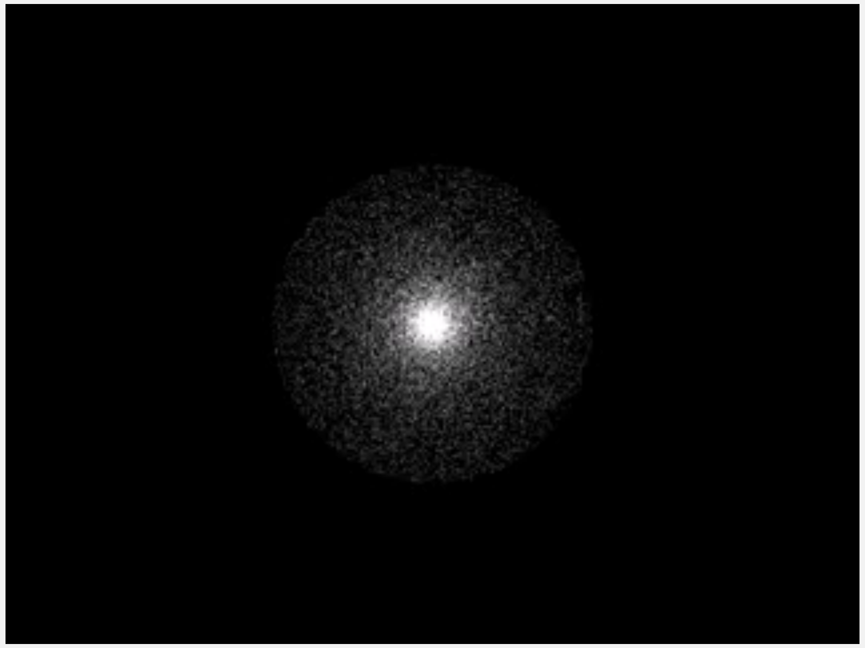
Some algorithmic successes

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





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

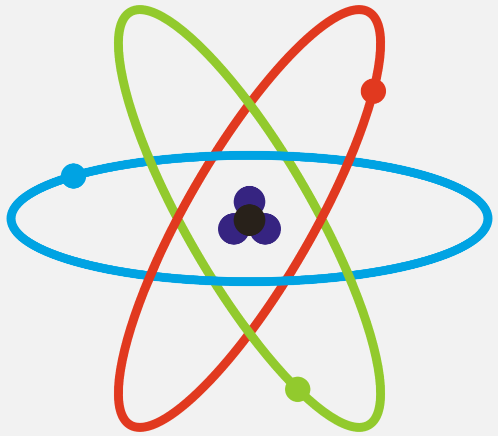
A framework for predicting performance and comparing algorithms.

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.

http://algs4.cs.princeton.edu

1.4 ANALYSIS OF ALGORITHMS

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

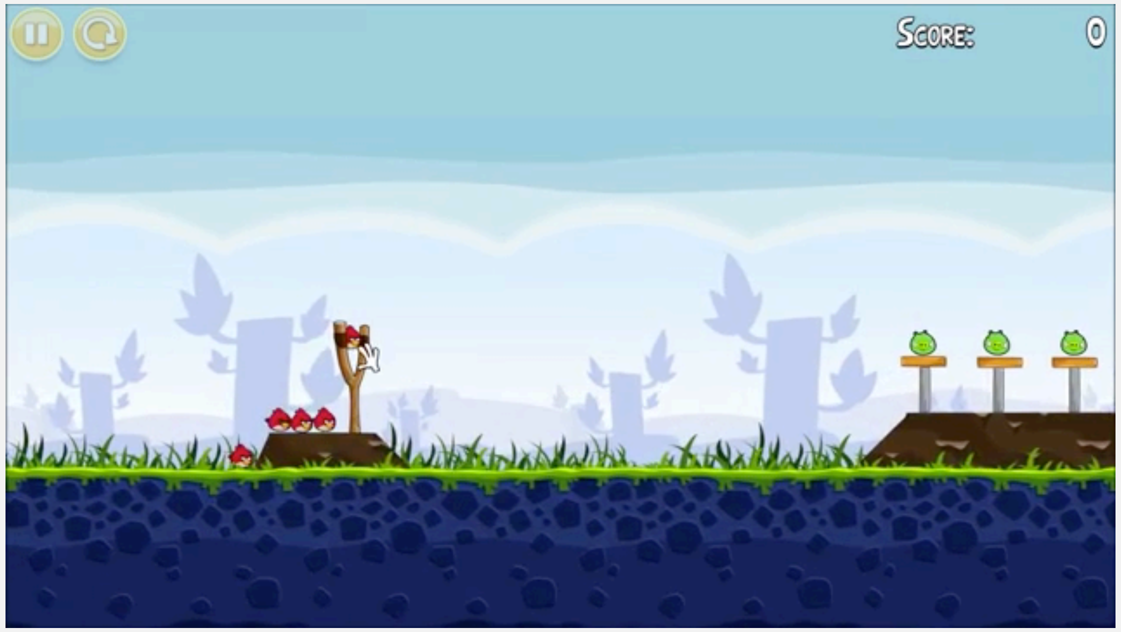
Example: 3-SUM

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)
                  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

for simplicity, ignore integer overflow

Measuring the running time

Q. How to time a program?

A. Manual.

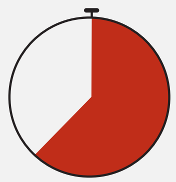

```
% java ThreeSum 1Kints.txt
```

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?

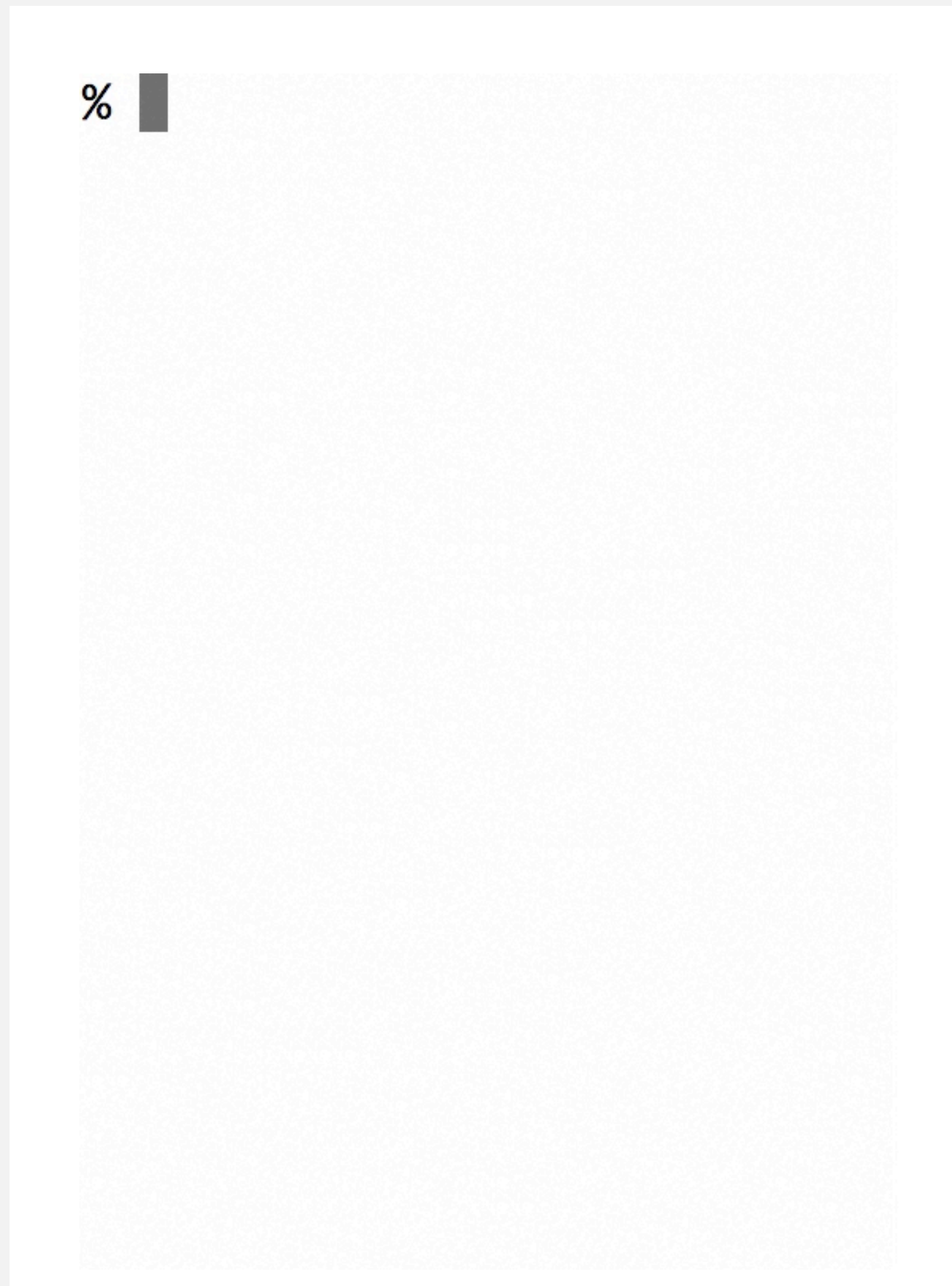
A. Automatic.

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.



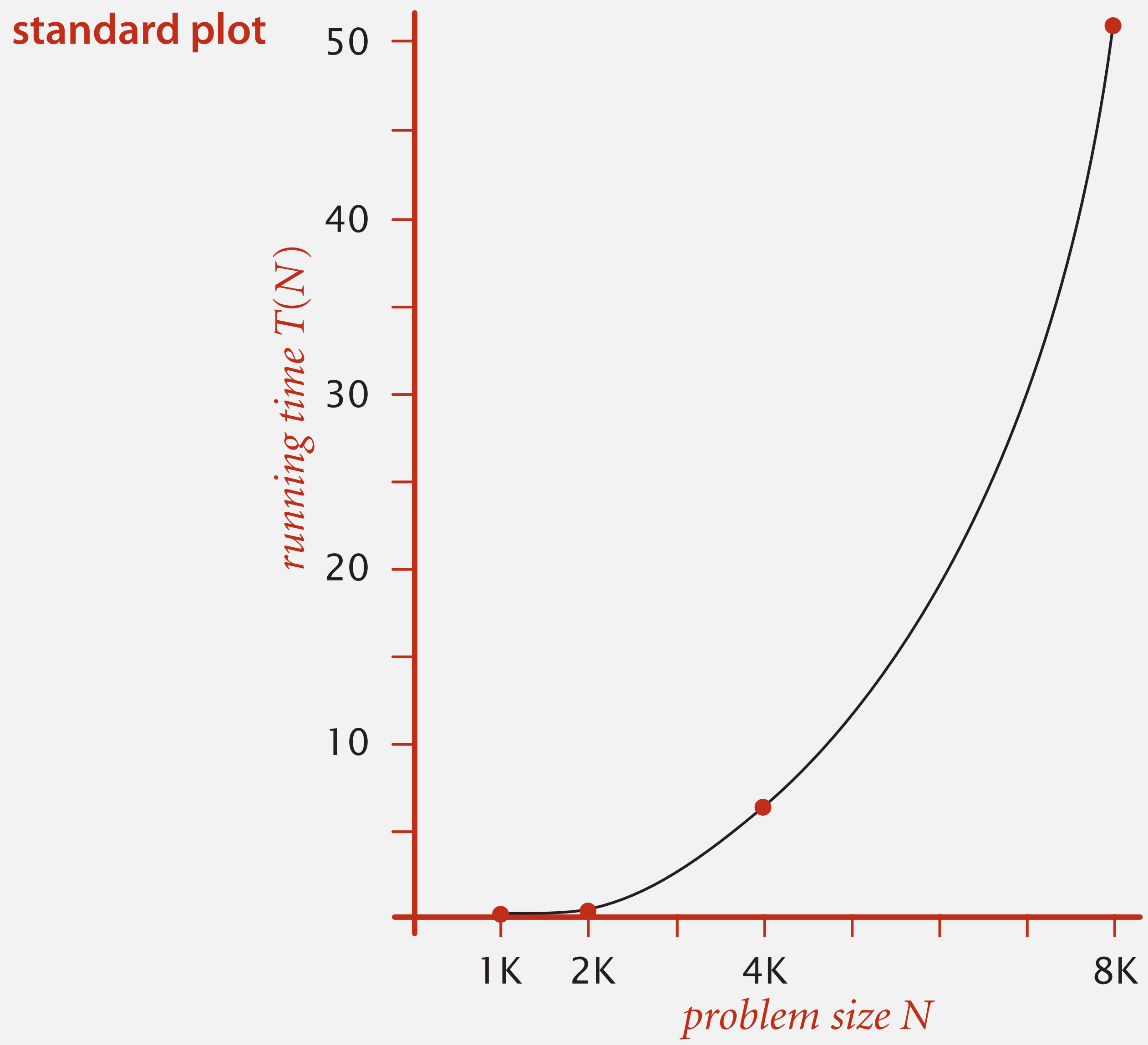
Empirical analysis

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	?

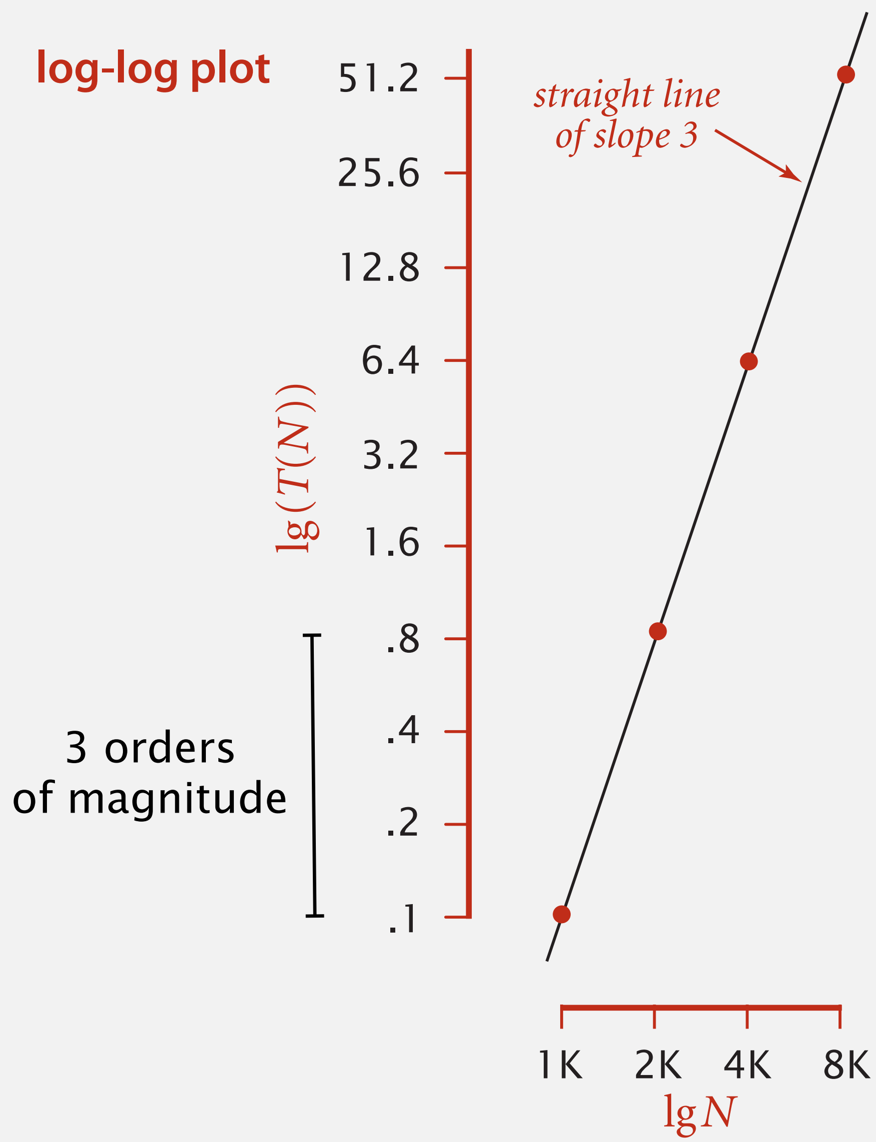
Data analysis

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.



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

$b = 2.999$

$c = -33.2103$

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

power law

slope

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.

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 N3 [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.

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

$$\log_2\left(\frac{T(2N)}{T(N)}\right) = b$$

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 (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

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

http://algs4.cs.princeton.edu

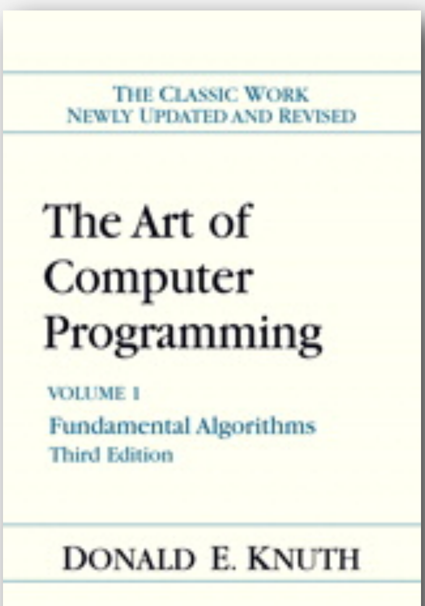
1.4 Analysis of Algorithms

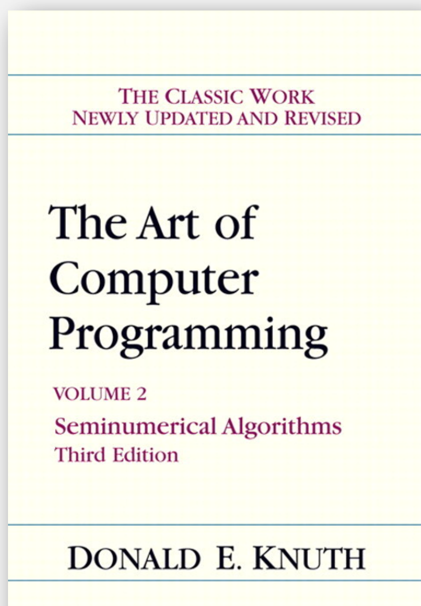
- introduction
- observations
- **mathematical models**
- order-of-growth classifications
- theory of algorithms
- memory

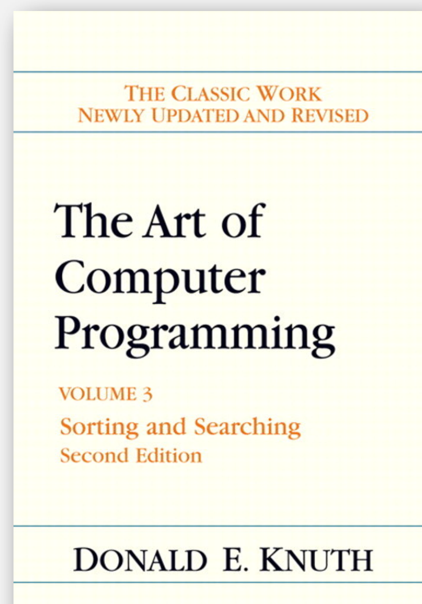
Mathematical models for running time

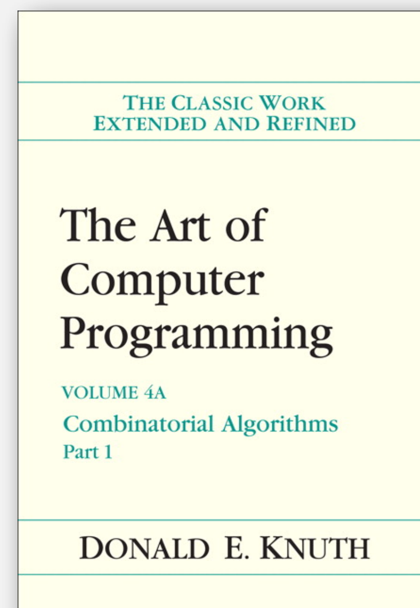
Total running time: sum of (cost × frequency) for all operations.

- 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.

novice mistake: abusive string concatenation

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 $2N$

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++;
```

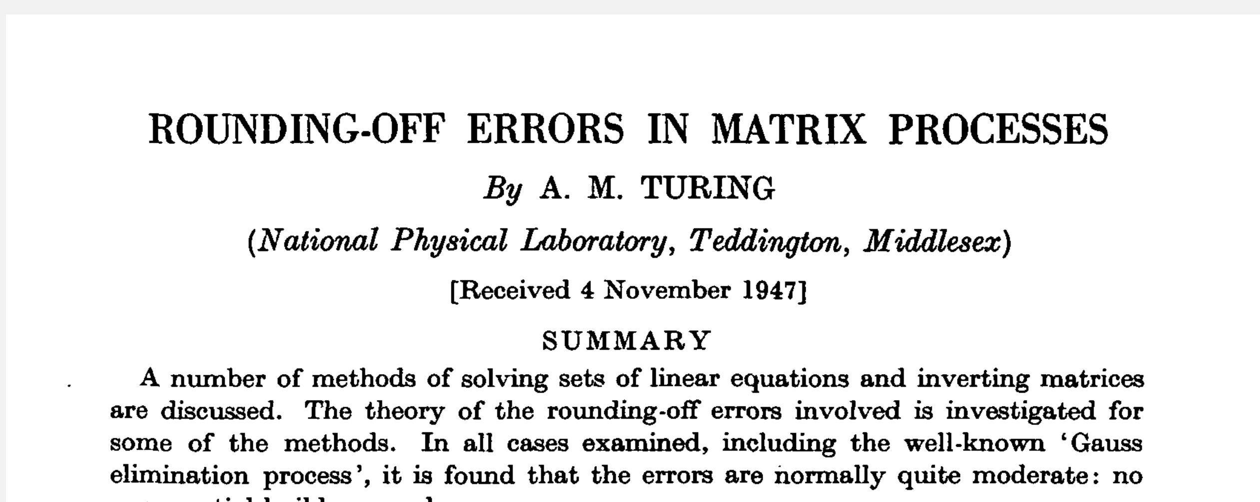
$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$½ (N + 1) (N + 2)$
equal to compare	$½ N (N - 1)$
array access	$N (N - 1)$
increment	$½ N (N - 1)$ 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

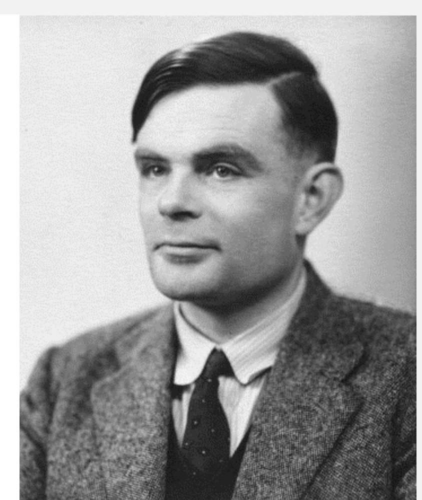
By A. M. TURING

(*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++;
```

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

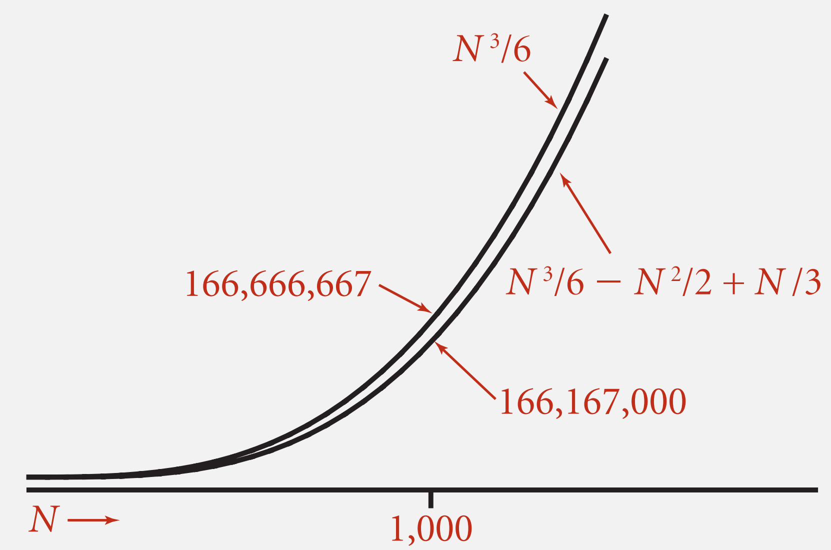
operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$½ (N + 1) (N + 2)$
equal to compare	$½ N (N - 1)$
array access	$N (N - 1)$
increment	$½ N (N - 1)$ to $N (N - 1)$

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



Leading-term approximation

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

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

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

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

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"

A. $\sim N^2$ array accesses, simplified from $N(N - 1) = N^2 - N$

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"

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

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

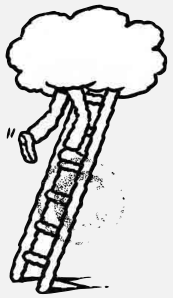
Bottom line. Use cost model and tilde notation to simplify counts.

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

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 3-SUM is N^3, simplified from $\sim\frac{1}{2} N^3$

3-SUM

```
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.

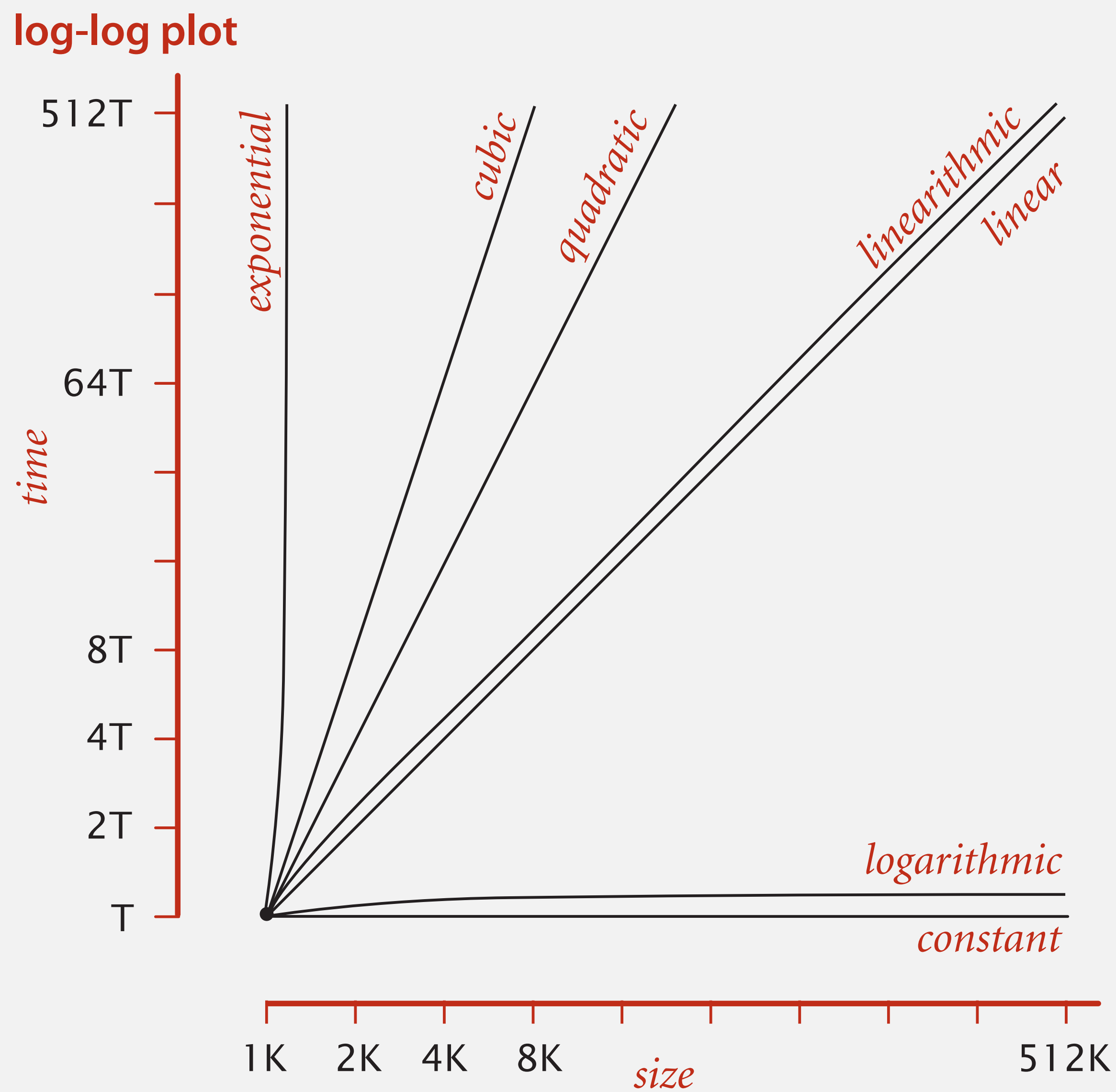
where leading coefficient depends on machine, compiler, JVM, ...

Common order-of-growth classifications

Good news. The set of functions

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

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



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**	[see combinatorial search lecture]	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 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.

```
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;
}
```

one "3-way compare"

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 ≤ N.

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

left or right half (floored division) — $T(N/2)$

possible to implement with one 2-way compare (instead of 3-way) — $+1$

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

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

START RECORDING

Attendance Quiz

Attendance Quiz: Order of Growth

- Scan the QR code, or find today's attendance quiz under the "Quizzes" tab on Canvas
- Password: to be announced in class

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	Constant	`a = b + c;`	statement	add two numbers	1
?	**?**	`for (int i = 0; i < N; i++)` `{ ... }`	loop	find the maximum	2
?	**?**	`while (N > 1)` `{ N = N / 2; ... }`	divide in half	binary search	~ 1

?	**?**	`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

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

Attendance Quiz: Order of Growth

- Write your name
- Fill in the “order of growth” and “name” columns

order of growth	name	typical code framework	description	example	$T(2N) / \mathrm{T}(N)$
1	Constant	`a = b + c;`	statement	add two numbers	1
?	**?**	`for (int i = 0; i < N; i++)` `{ ... }`	loop	find the maximum	2
?	**?**	`while (N > 1)` `{ N = N / 2; ... }`	divide in half	binary search	~ 1

?	**?**	`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

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

Binary Search Demo

Last Lecture…

- We saw a black-box approach for measuring the growth rate of an algorithm (i.e., doubling analysis)
- We saw a code analysis approach for determining the growth rate of an algorithm (i.e., counting operations)
- This lecture: is an algorithm's growth rate optimal? Or could a better algorithm exist?

http://algs4.cs.princeton.edu

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.

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: $\sim \lg N$

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 Oh	$\Theta(N^2)$ 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: Fibonacci example

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. FIBONACCI = "What is the N^{th} element in the sequence?"

Upper bound. A specific algorithm.

- Ex. Recursive algorithm for FIBONACCI
- Running time of the optimal algorithm for FIBONACCI is $O(2^N)$.

Theory of algorithms: Fibonacci example

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. FIBONACCI = "What is the N^{th} element in the sequence?"

Upper bound. A specific algorithm.

- Ex. Improved dynamic algorithm for FIBONACCI.
- Running time of the optimal algorithm for FIBONACCI is $O(N)$.

Lower bound. Proof that no algorithm can do better.

- Do we need to compute the sequence, or is there a closed-form solution (i.e., can we jump right to the answer with a formula)? Yes!
- Running time of the optimal algorithm for FIBONACCI is $\Omega(1)$.

Theory of algorithms: Fibonacci example

Goals.

- Establish "difficulty" of a problem and develop "optimal" algorithms.
- Ex. FIBONACCI = "What is the N^{th} element in the sequence?"

Upper bound. A specific algorithm.

- Ex. Improved closed-form solution for FIBONACCI
- Running time of the optimal algorithm for FIBONACCI is $O(1)$.

Lower bound. Proof that no algorithm can do better.

- Do we need to compute the sequence, or is there a closed-form solution (i.e., can we jump right to the answer with a formula)? Yes!
- Running time of the optimal algorithm for FIBONACCI is $\Omega(1)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Closed form for FIBONACCI is optimal: running time is $\Theta(1)$.

Theory of algorithms: 1-SUM example

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: 3-SUM example

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: 3-SUM example

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 $O(N^2 \log N)$.

An $N^2 \log N$ 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^3

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, $N^2\log(N)$

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

Theory of algorithms: 3-SUM example

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?

Big O: The upper bound

$$f(n) \in O\left(g(n)\right) \text{ as } n \to \infty$$

when $f(n) \leq M \cdot g(n)$ for all $n \geq$ some n_o

$5n + 5 \in O(n^2)?$	Yes
$5n + 5 \in O(n)?$	Yes
$5n + 5 \in O(\log(n))?$	No
$5n + 5 \in O(1)?$	No

Big Ω: The lower bound

$$f(n) \in \Omega\big(g(n)\big) \text{ as } n \to \infty$$

$$\text{when } f(n) \geq M \cdot g(n) \text{ for all } n \geq \text{ some } n_o$$

$5n + 5 \in \Omega(n^2)?$	No
$5n + 5 \in \Omega(n)?$	Yes
$5n + 5 \in \Omega(\log(n))?$	Yes
$5n + 5 \in \Omega(1)?$	Yes

Big Θ: The tight bound

$$f(n) \in \Theta\big(g(n)\big) \text{ when } f(n) \in O\big(g(n)\big) \text{ and } f(n) \in \Omega\big(g(n)\big)$$

$5n + 5 \in \Theta(n^2)?$	No
$5n + 5 \in \Theta(n)?$	Yes
$5n + 5 \in \Theta(\log(n))?$	No
$5n + 5 \in \Theta(1)?$	No

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, conflating big-Oh and big-Theta.

In a job interview: “What is the big-Oh of this algorithm?”

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

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

Basics

Bit. 0 or 1.

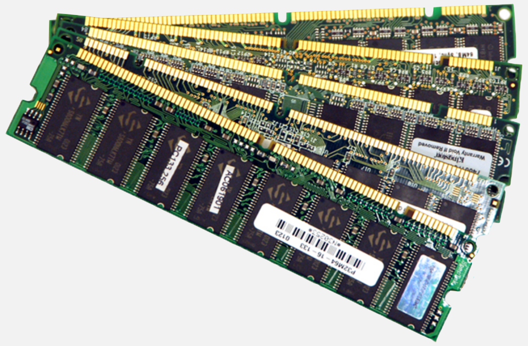
Byte. 8 bits.

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.

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

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[][]`	$\sim 2\,M\,N$
`int[][]`	$\sim 4\,M\,N$
`double[][]`	$\sim 8\,M\,N$

two-dimensional arrays

Typical memory usage for objects in Java

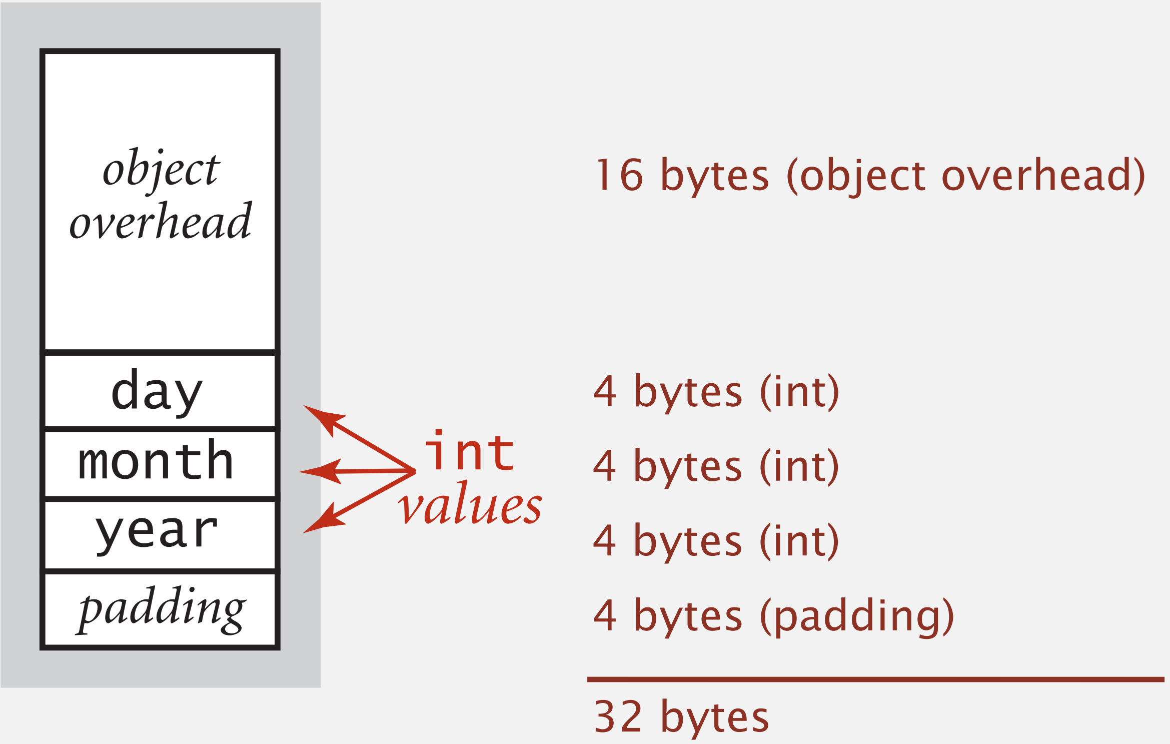
Object overhead. 16 bytes.

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)

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, count memory (recursively) for referenced object.

Memory profiler

Classmexer library. Measure memory usage by querying JVM.
http://www.javamex.com/classmexer

```
import com.javamex.classmexer.MemoryUtil;

public class Memory {
   public static void main(String[] args) {
      Date date = new Date(12, 31, 1999);
      StdOut.println(MemoryUtil.memoryUsageOf(date));
      String s = "Hello, World";
      StdOut.println(MemoryUtil.memoryUsageOf(s));        ← shallow
      StdOut.println(MemoryUtil.deepMemoryUsageOf(s));    ← deep
   }
}
```

```
% javac -cp .:classmexer.jar Memory.java
% java  -cp .:classmexer.jar -javaagent:classmexer.jar Memory
32
40    ← don't count char[]
88    ← 2N + 64
```

use **-XX:-UseCompressedOops** on OS X to match our model

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.

Memory Over Time

Year Introduced	Model	Default Memory
1984	Original Apple Macintosh	128 KB
1986	Macintosh Plus	1 MB
1990	Macintosh LC	2MB
1998	iMac G3	32MB
2002	iMac G4	128 MB
2004	iMac G4	256 MB
2006	Intel iMac	512 MB
2008	Intel iMac	1 GB
2010	Intel iMac	4 GB
2012	Intel iMac	8 GB
2014	Intel iMac	8 GB
2017	Intel iMac	8 GB
2019	Intel iMac	8 GB
2021	M1 iMac	8 GB
2023	M2 Mac Mini	8 GB

http://www.allaboutapple.com/

David Fuchs and Rama, Wikimedia

https://apple-history.com/imac_coreduo

https://www.apple.com

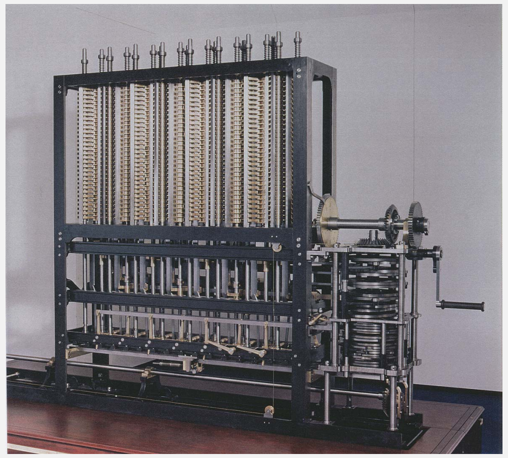
Turning the crank: summary

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.

Comparing Arrays to other Data Structures

- Using an array, what is the time complexity of:
  - Updating an element, given the index?
  - Growing the length of the array?
  - Locate an element (sorted or unsorted)?
- We'll learn about data structures that:
  - Grow with O(1) – linked lists, stacks, queues, and dictionaries (AKA, maps, symbol tables)
  - Locate an element with O(1) – dictionaries

If time:
Performance Slides