INTRODUCTION TO
COMPUTER SCIENCE
Rutgers University

11. Performance

http://introcs.cs.rutgers.edu

INTRODUCTION TO
COMPUTER SCIENCE
Rutgers University

11. Performance

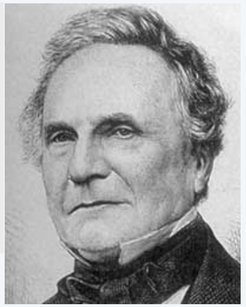
- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

Adopted and modified from Slides by Sedgewick and Wayne

The challenge (since the earliest days of computing machines)

" 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?"

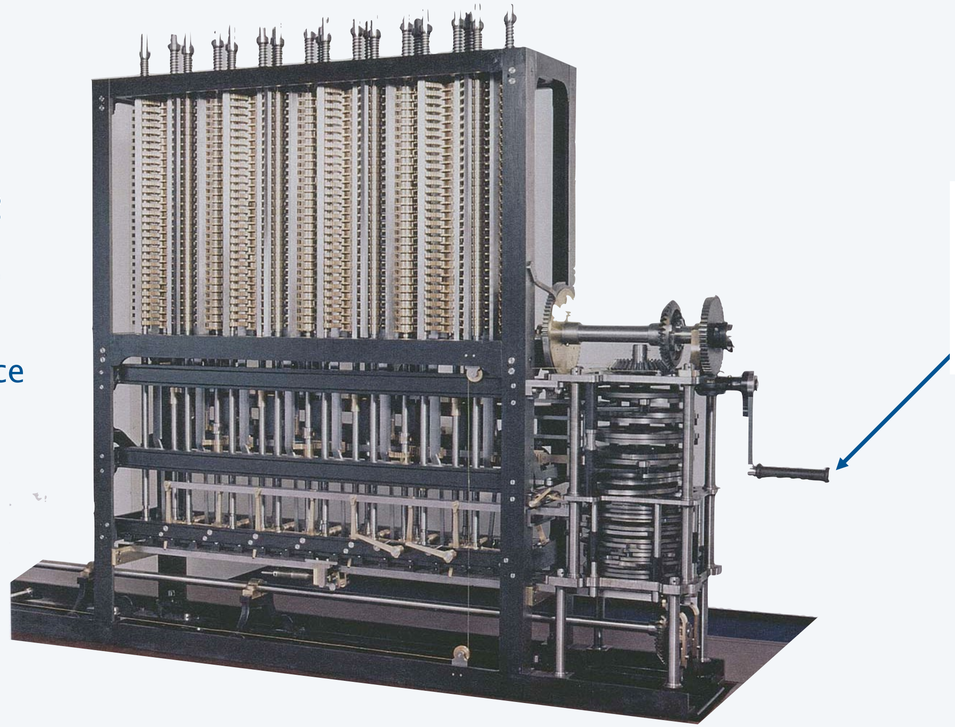
– Charles Babbage



Difference Engine #2

Designed by Charles Babbage, c. 1848

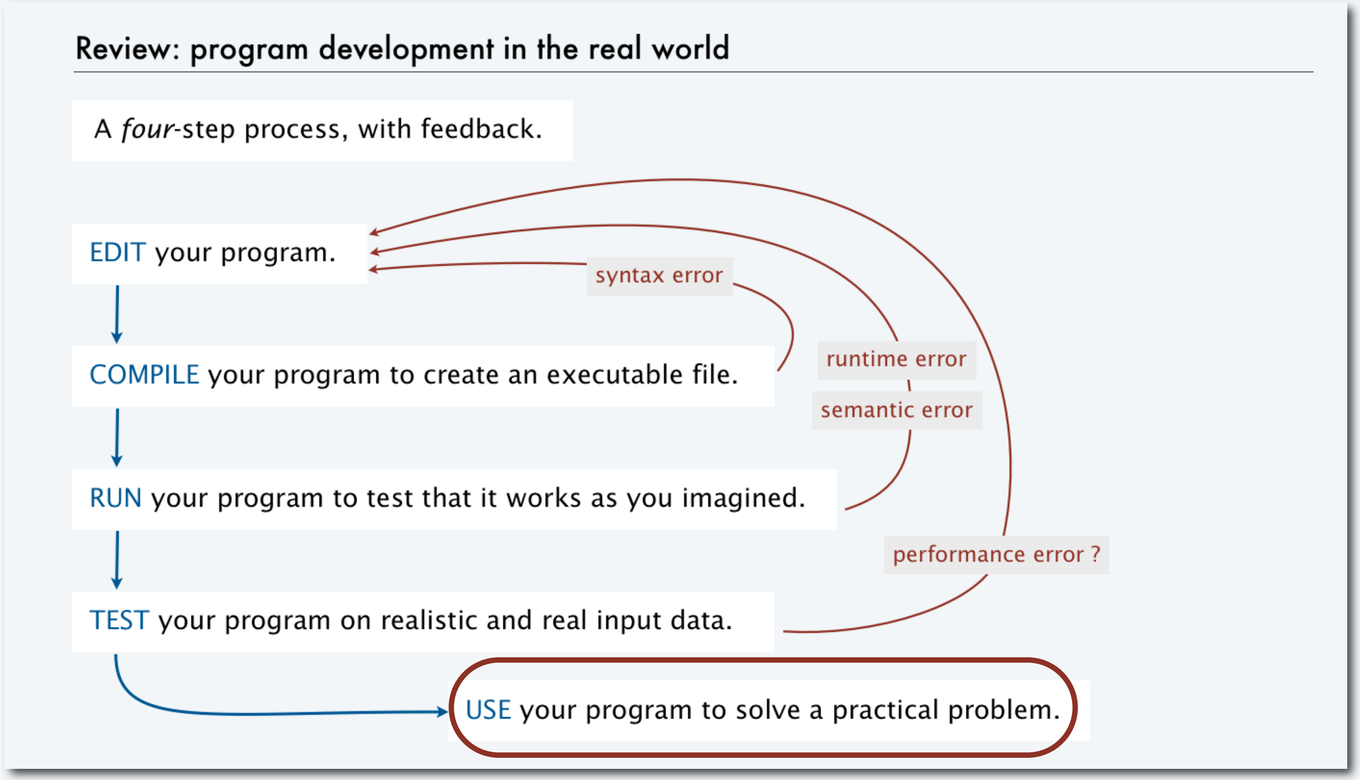
Built by London Science Museum, 1991



Q. How many times do you have to turn the crank?

The challenge (modern version)

Q. Will I be able to use my program to solve a large practical problem?



Q. If not, how might I understand its performance characteristics so as to improve it?

Key insight (Knuth 1970s). Use the *scientific method* to understand performance.

Three reasons to study program performance

1. To predict program behavior

- Will my program finish?
- *When* will my program finish?

2. To compare algorithms and implementations.

- Will this change make my program faster?
- How can I make my program faster?

3. To develop a basis for understanding the problem and for designing new algorithms

- Enables new technology.
- Enables new research.

```
public class Gambler
{
   public static void main(String[] args)
   {
     int stake  = Integer.parseInt(args[0]);
     int goal   = Integer.parseInt(args[1]);
     int trials = Integer.parseInt(args[2]);
     int wins   = 0;
     for (int t = 0; t < trials; t++)
     {
        int cash = stake;
        while (cash > 0 && cash < goal)
           if (Math.random() < 0.5) cash++;
           else                     cash--;
        if (cash == goal) wins++;
     }
     StdOut.print(wins + " wins of " + trials);
  }
}
```

An *algorithm* is a method for solving a problem that is suitable for implementation as a computer program.

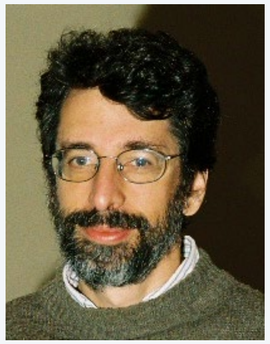
We study several algorithms later in this course.

Taking more CS courses? You'll learn dozens of algorithms.

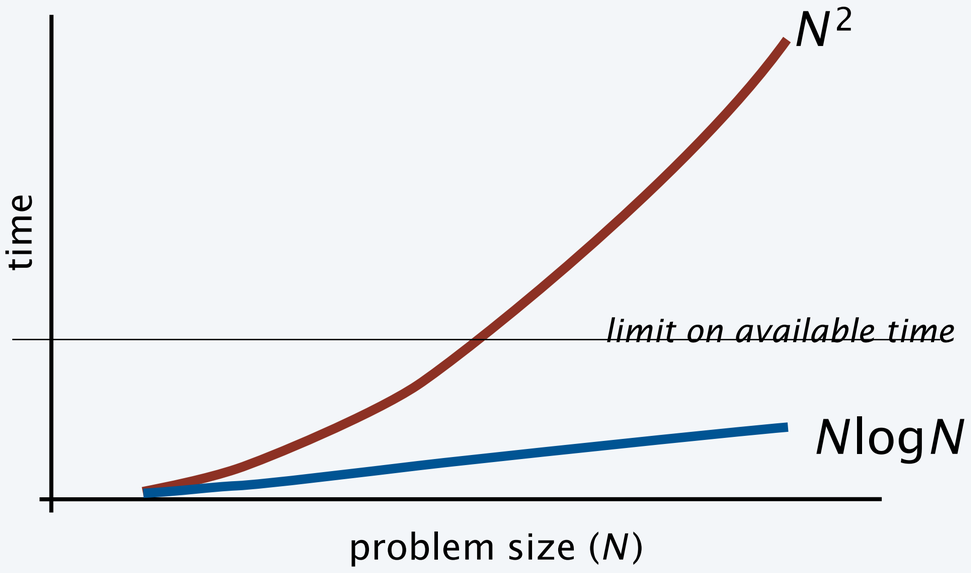
An algorithm design success story

N-body simulation

- Goal: Simulate gravitational interactions among N bodies.
- Brute-force algorithm uses N^2 steps per time unit.
- Issue (1970s): Too slow to address scientific problems of interest.
- Success story: *Barnes-Hut* algorithm uses $N \log N$ steps and *enables new research*.



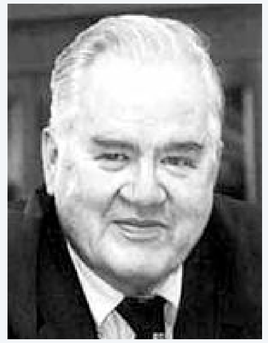
Andrew Appel
Princeton '81
senior thesis



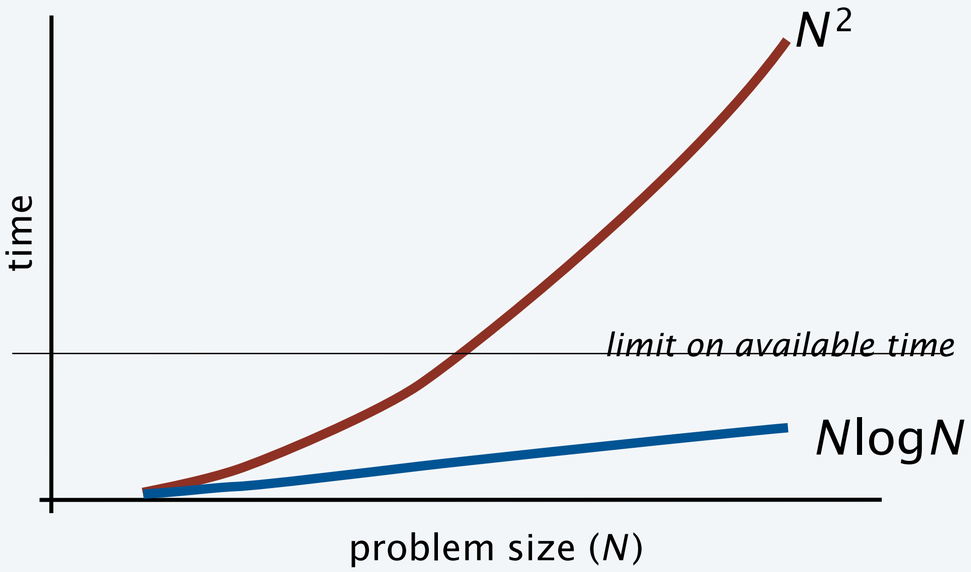
Another algorithm design success story

Discrete Fourier transform

- Goal: Break down waveform of N samples into periodic components.
- Applications: digital signal processing, spectroscopy, ...
- Brute-force algorithm uses N^2 steps.
- Issue (1950s): Too slow to address commercial applications of interest.
- Success story: *FFT* algorithm uses $N \log N$ steps and *enables new technology*.



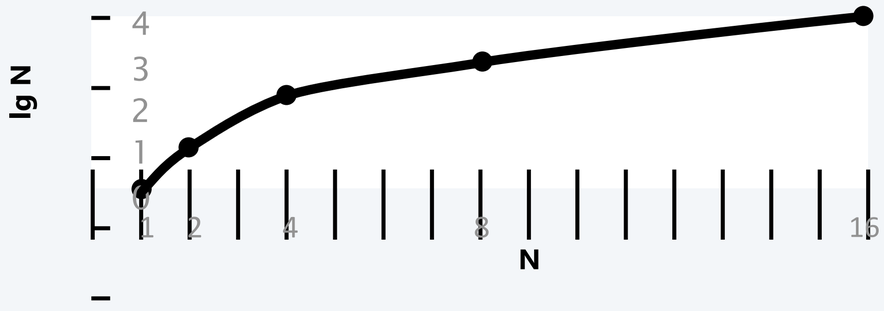
John Tukey
1915–2000



Quick aside: binary logarithms

Def. The *binary logarithm* of a number N (written lg N) is the number x satisfying $2^x = N$.

or $\log_2 N$



Q. How many recursive calls for convert(N)?

```
public static String convert(int N)
{
   if (N == 1) return "1";
   return convert(N/2) + (N % 2);
}
```

Frequently encountered values

N	*approximate value*	lgN	$\log_{10}$N
2^{10}	1 thousand	10	3.01
2^{20}	1 million	20	6.02
2^{30}	1 billion	30	9.03

A. Largest integer less than or equal to lg N (written $\lfloor \lg N \rfloor$).

← Prove by induction.
Details in "sorting and searching" lecture.

Fact. The number of bits in the binary representation of N is $1+\lfloor \lg N \rfloor$.

Fact. Binary logarithms arise in the study of algorithms based on recursively solving problems half the size (*divide-and-conquer algorithms*), like convert, FFT and Barnes-Hut.

Performance or Efficiency of Algorithms

Efficiency is a measure of speed and space consumption

Speed – time complexity

- also called running or execution time

Space consumption – space complexity

- memory usage
- always *less than or equal to* the time requirement

An algorithmic challenge: 3-sum problem

Three-sum. Given *N* integers, enumerate the triples that sum to 0.

For simplicity, just count them.

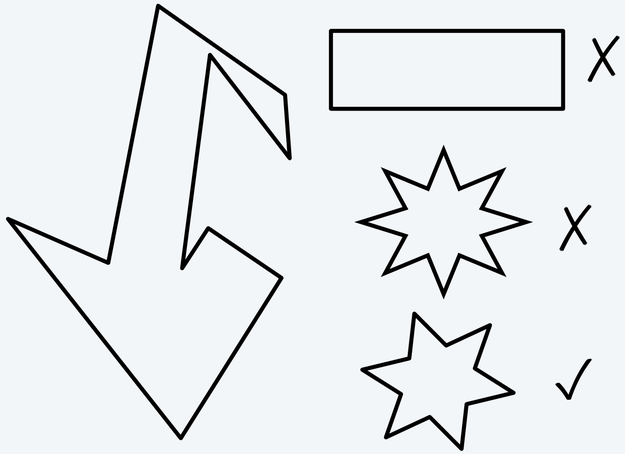
```
public class ThreeSum
{
   public static int count(int[] a)
   {  /* See next slide. */ }

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

```
% more 6ints.txt

30 -30 -20 -10 40  0

% java ThreeSum <  6ints.txt
3
```

30	-30	0
30	-20	-10
-30	-10	40

Applications in computational geometry

- Find collinear points.
- Does one polygon fit inside another?
- Robot motion planning.
- [a surprisingly long list]



Q. Can we solve this problem for *N* = 1 million?

Three-sum implementation

"Brute force" algorithm

- Process all possible triples.
- Increment counter when sum is 0.

i	0	1	2	3	4	5
a[i]	30	-30	-20	-10	40	0

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 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)
               cnt++;
   return cnt;
}
```

Keep i < j < k to avoid processing each triple 6 times

$\binom{N}{3}$ triples with i < j < k

i	j	k	a[i]	a[j]	a[k]
0	1	2	30	-30	-20
0	1	3	30	-30	-10
0	1	4	30	-30	40
0	1	5	30	-30	0
0	2	3	30	-20	-10
0	2	4	30	-20	40
0	2	5	30	-20	0
0	3	4	30	-10	40
0	3	5	30	-10	0
0	4	5	30	40	0
1	2	3	-30	-20	-10
1	2	4	-30	-20	40
1	2	5	-30	-20	0
1	3	4	-30	-10	40

Q. How much time will this program take for $N = 1$ million?

An algorithmic challenge: 3-sum problem (with timing)

Explain algorithmic efficiency as it relates to speed and space consumption.

LO 11.1a

Three-sum. Given *N* random integers, enumerate the triples that sum to 0.

```
public class ThreeSum
{
   public static int count(int[] a)
   {  /* See previous slide. */ }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int[] a = new int[N];
      Random r = new Random();
      for (int i=0; i< N; i++)
          a[i] = r.nextInt()
      StdOut.println(count(a));
   }
}
```

```
% java ThreeSum  10000
```

N	time
10	
100	
1000	
10000	
1000000	
10^9	??
10^{12}	??
10^{15}	??

Q. Complete the table for N=10, 100, 1000, …

Memory usage by a program

Explain algorithmic efficiency as it relates to speed and space consumption.

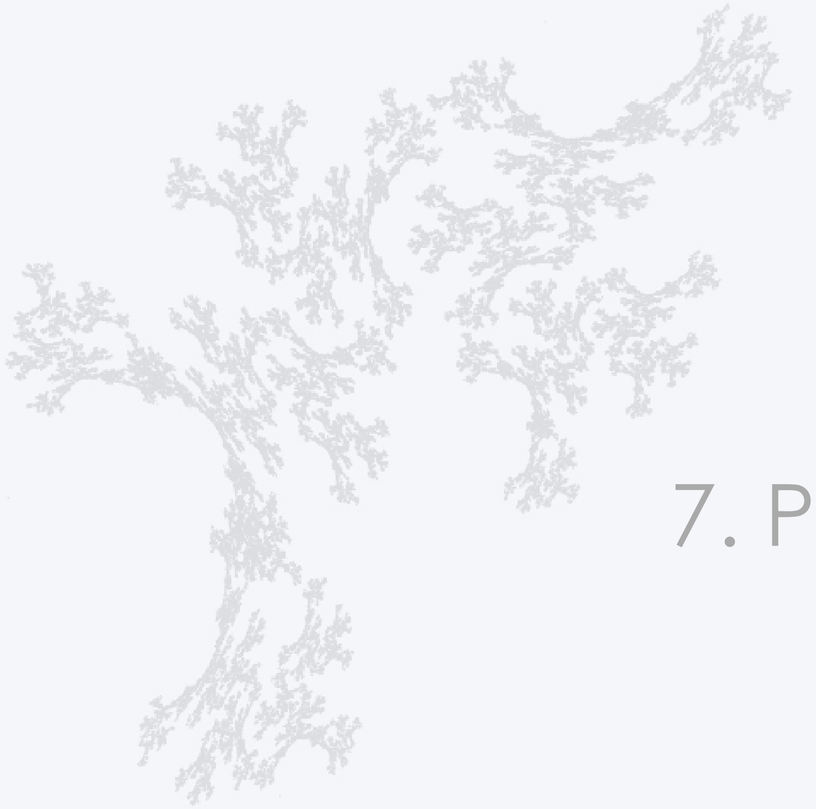
LO 11.1a

Memory. It is possible to count how much memory might be used by a program.

```
public static int[ ] reverse(int[ ] a) {
    int n = a.length;
    for (int i=0; i< a.length/2; i++) {
        int tmp = a[i];
        a[i] = a[n-i-1];
        a[n-i-1] = tmp;
    }
    return a;
}
```

Q. Assuming ints are 4 bytes, what is the total memory requirement for reverse function?

A. $4n + 4 + 4 + 4 = 4n + 12$



7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

A first step in analyzing running time

Find representative inputs

- Option 1: Collect actual input data.
- Option 2: Write a program to generate representative inputs.

Input generator for ThreeSum

```
public class Generator
{  // Generate N integers in [-M, M)
   public static void main(String[] args)
   {
      int M = Integer.parseInt(args[0]);
      int N = Integer.parseInt(args[1]);
      for (int i = 0; i < N; i++)
         StdOut.println(StdRandom.uniform(-M, M));
   }
}
```

```
% java Generator 1000000 10
28773
-807569
-425582
594752
600579
-483784
-861312
-690436
-732636
360294
```

not much chance of a 3-sum

```
% java Generator 10 10
-2
1
-4
1
-2
-10
-4
1
0
-7
```

good chance of a 3-sum

Empirical analysis

Run experiments

- Start with a moderate input size *N*.
- Measure and record running time.
- Double input size *N*.
- Repeat.
- Tabulate and plot results.

Measure running time

Replace println() in ThreeSum with this code.

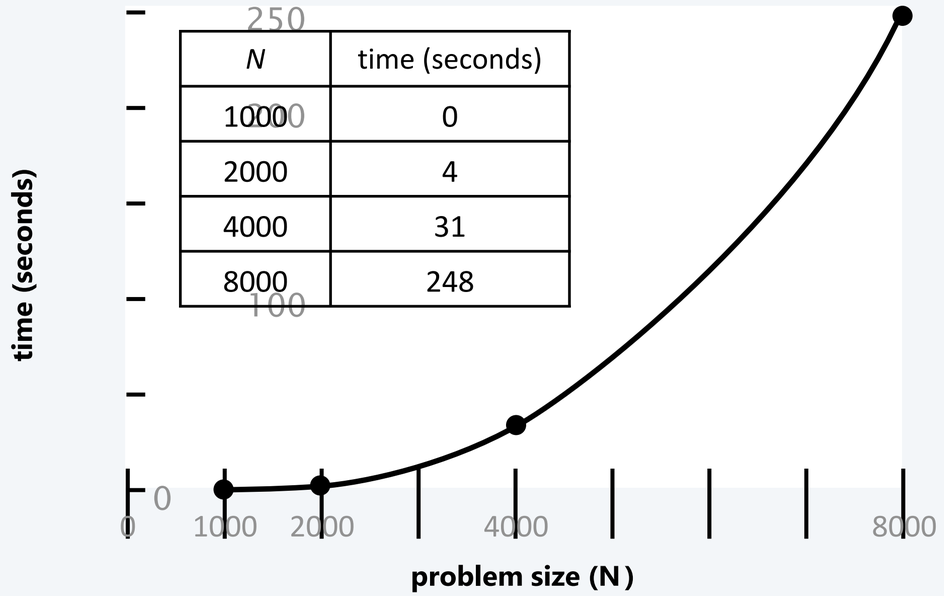
```
double start = System.currentTimeMillis() / 1000.0;
int cnt = count(a);
double now   = System.currentTimeMillis() / 1000.0;
StdOut.printf("%d (%.0f seconds)\n", cnt, now - start);
```

Run experiments

```
% java Generator 1000000 1000 | java ThreeSum
59 (0 seconds)
% java Generator 1000000 2000 | java ThreeSum
522 (4 seconds)
% java Generator 1000000 4000 | java ThreeSum
3992 (31 seconds)
% java Generator 1000000 8000 | java ThreeSum
31903 (248 seconds)
```

Tabulate and plot results

N	time (seconds)
1000	0
2000	4
4000	31
8000	248



Aside: experimentation in CS

is *virtually free*, particularly by comparison with other sciences.

Chemistry

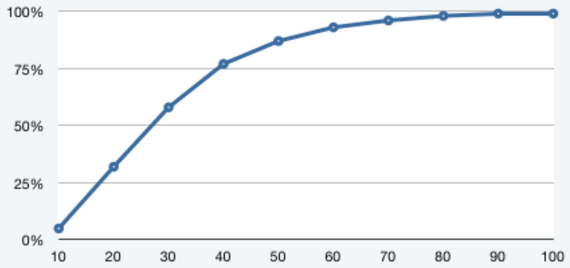
Biology

one experiment

Physics

one million experiments

```
% java SelfAvoidingWalker 10 100000
5% dead ends
% java SelfAvoidingWalker 20 100000
32% dead ends
% java SelfAvoidingWalker 30 100000
58% dead ends
% java SelfAvoidingWalker 40 100000
77% dead ends
% java SelfAvoidingWalker 50 100000
87% dead ends
% java SelfAvoidingWalker 60 100000
93% dead ends
% java SelfAvoidingWalker 70 100000
96% dead ends
% java SelfAvoidingWalker 80 100000
98% dead ends
% java SelfAvoidingWalker 90 100000
99% dead ends
% java SelfAvoidingWalker 100 100000
99% dead ends
```



Computer Science

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

Data analysis

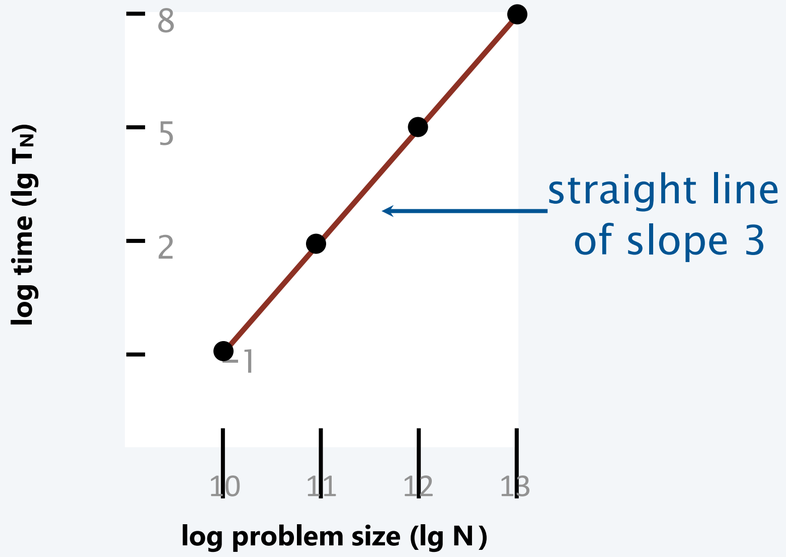
Curve fitting

- Plot on *log-log scale*.
- If points are on a straight line (often the case), a *power law* holds—a curve of the form aN^b fits.
- The exponent b is the slope of the line.
- Solve for a with the data.

N	T_N	lgN	lg T_N	$4.84 \times 10^{-10} \times N^3$
1000	0.5	10	−1	0.5
2000	4	11	2	4
4000	31	12	5	31
8000	248	13	8	248

✓

log-log plot



Do the math

x-intercept (use lg in anticipation of next step)

$$\lg T_N = \lg a + 3 \lg N$$ equation for straight line of slope 3

$$T_N = aN^3$$ raise 2 to a power of both sides

$$248 = a \times 8000^3$$ substitute values from experiment

$$a = 4.84 \times 10^{-10}$$ solve for a

$$T_N = 4.84 \times 10^{-10} \times N^3$$ substitute

a curve that fits the data ?

Prediction and verification

Hypothesis. Running time of ThreeSum is $4.84 \times 10^{-10} \times N^3$.

Prediction. Running time for $N = 16{,}000$ will be 1982 seconds.

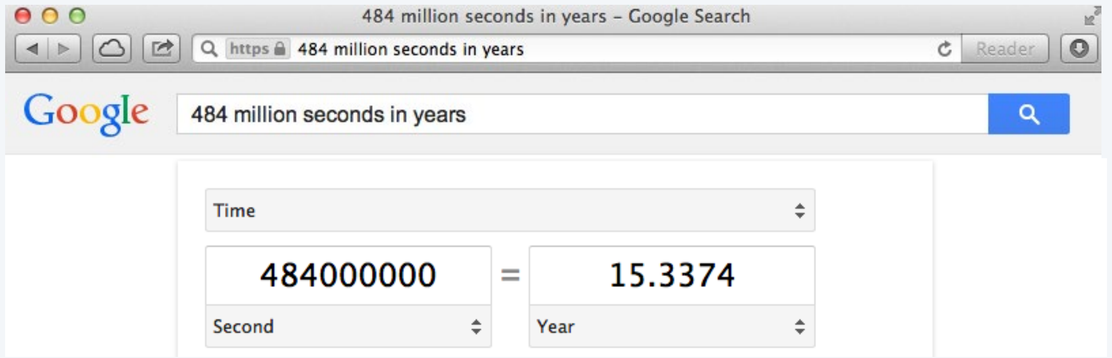
about half an hour

```
% java Generator 1000000 16000 | java ThreeSum
31903 (1985 seconds)
```

✓

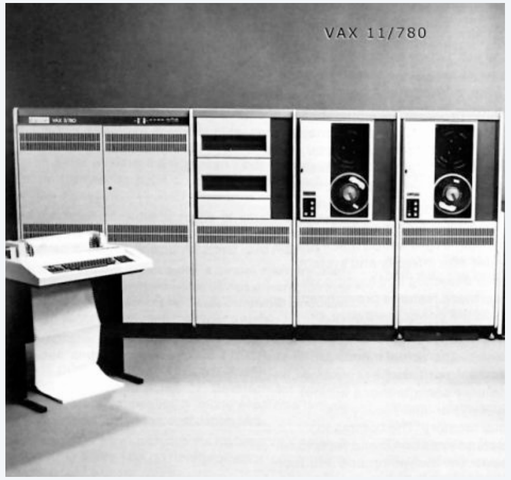
Q. How much time will this program take for $N = 1$ million?

A. 484 million seconds (more than 15 years).



Another hypothesis

1970s

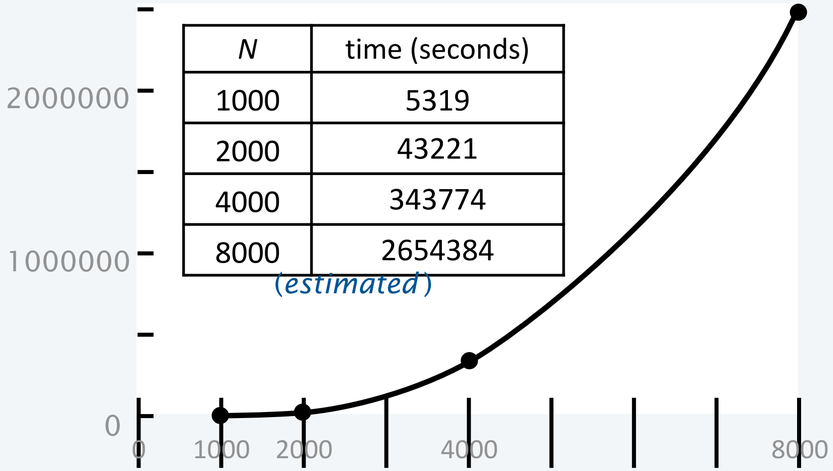


VAX 11/780

$5.2 \times 10^{-6} \times N^3$ seconds

N	time (seconds)
1000	5319
2000	43221
4000	343774
8000	2654384

(estimated)

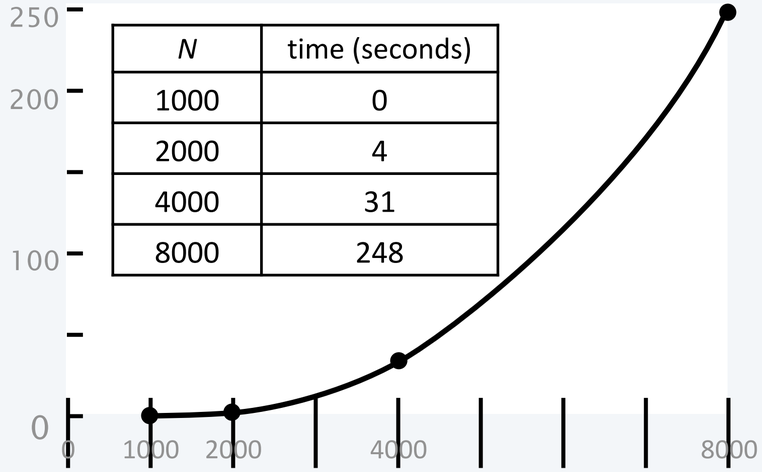


2010s: 10,000+ times faster

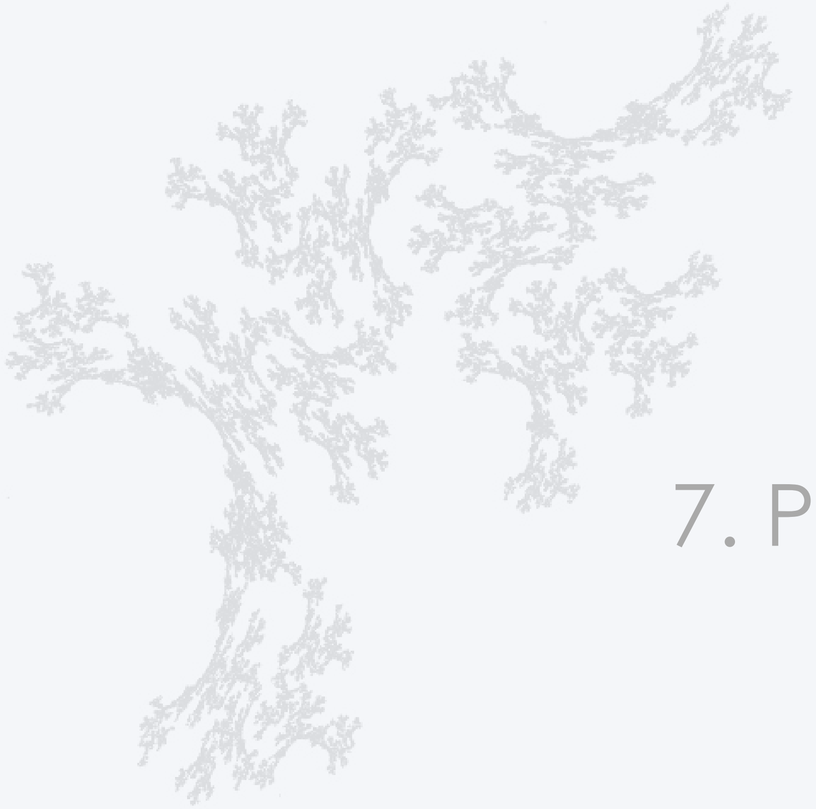
Macbook Air

$4.84 \times 10^{-10} \times N^3$ seconds

N	time (seconds)
1000	0
2000	4
4000	31
8000	248



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



7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

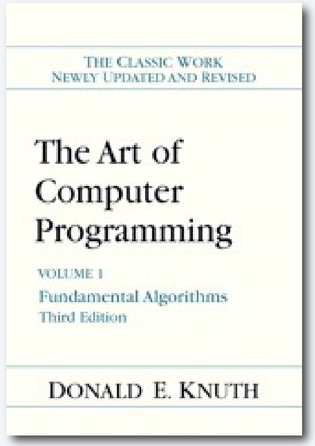
Mathematical models for running time

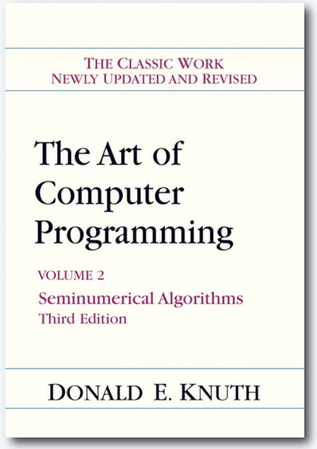
Q. Can we write down an accurate formula for the running time of a computer program?

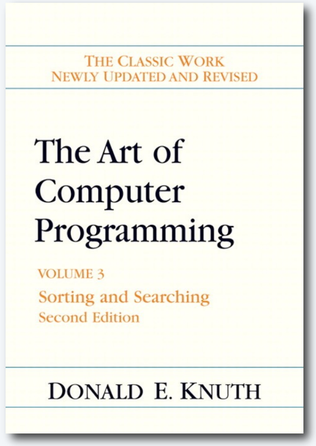
A. (Prevailing wisdom, 1960s) No, too complicated.

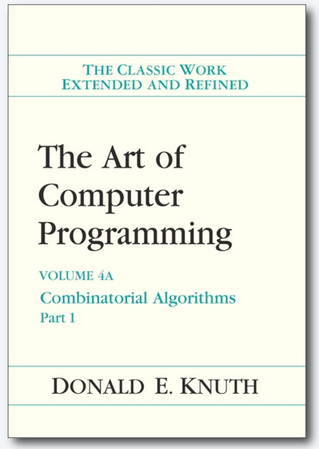
A. (D. E. Knuth, 1968–*present*) Yes!

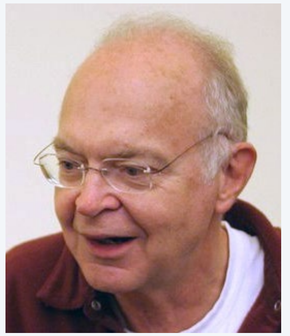
- Determine the set of operations.
- Find the *cost* of each operation (depends on computer and system software).
- Find the *frequency of execution* of each operation (depends on algorithm and inputs).
- Total running time: sum of cost × frequency for all operations.











Don Knuth
1974 Turing Award

Warmup: 1-sum

Identify the basic operations in an algorithm and determine the running time of an algorithm by counting its basic operations.

LO 11.1c

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 0;
   for (int i = 0; i < N; i++)
      if (a[i] == 0)
         cnt++;
   return cnt;
}
```

Note that frequency of increments depends on input.

operation	*cost*	*frequency*
function call/return	20 *ns*	1
variable declaration	2 *ns*	2
assignment	1 *ns*	2
less than compare	1/2 *ns*	$N + 1$
equal to compare	1/2 *ns*	N
array access	1/2 *ns*	N
increment	1/2 *ns*	between N and $2N$

representative estimates (with some poetic license); knowing exact values may require study and experimentation.

Q. Formula for total running time ?

A. $cN + 26.5$ nanoseconds, where c is between 2 and 2.5, depending on input.

Warmup: 2-sum

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 0;
   for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
         if (a[i] + a[j] == 0)
            cnt++;
   return cnt;
}
```

operation	*cost*	*frequency*
function call/return	20 *ns*	1
variable declaration	2 *ns*	$N + 2$
assignment	1 *ns*	$N + 2$
less than compare	1/2 *ns*	$(N + 1)\ (N + 2)/2$
equal to compare	1/2 *ns*	$N\ (N - 1)/2$
array access	1/2 *ns*	$N\ (N - 1)$
increment	1/2 *ns*	between $N\ (N + 1)/2$ and N^2

exact counts tedious to derive

i < j = $\binom{N}{2} = \frac{N(N-1)}{2}$

Q. Formula for total running time ?

A. $c_1 N^2 + c_2 N + c_3$ nanoseconds, where... [complicated definitions].

Simplifying the calculations

Tilde notation

- Use only the fastest-growing term.
- Ignore the slower-growing terms.

Rationale

- When N is large, ignored terms are negligible.
- When N is small, *everything* is negligible.

Def. $f(N) \sim g(N)$ means $f(N)/g(N) \to 1$ as $N \to \infty$

Ex. $5/4\ N^2 + 13/4\ N + 53/2 \sim 5/4\ N^2$

1,253,276.5 for $N = 1{,}000$

1,250,000 for $N = 1{,}000$, within .3%

Q. Formula for 2-sum running time when count is not large (typical case)?

eliminate dependence on input

A. $\sim 5/4\ N^2$ nanoseconds.

Mathematical model for 3-sum

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 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)
               cnt++;
   return cnt;
}
```

operation	*cost*	*frequency*
function call/return	20 *ns*	1
variable declaration	2 *ns*	~ N
assignment	1 *ns*	~ N
less than compare	1/2 *ns*	~ $N^3/6$
equal to compare	1/2 *ns*	~ $N^3/6$
array access	1/2 *ns*	~ $N^3/2$
increment	1/2 *ns*	~ $N^3/6$

assumes count is not large

i < j < k = $\binom{N}{3} = \frac{N(N-1)(N-2)}{6} \sim \frac{N^3}{6}$

Q. Formula for total running time when return value is not large (typical case)?

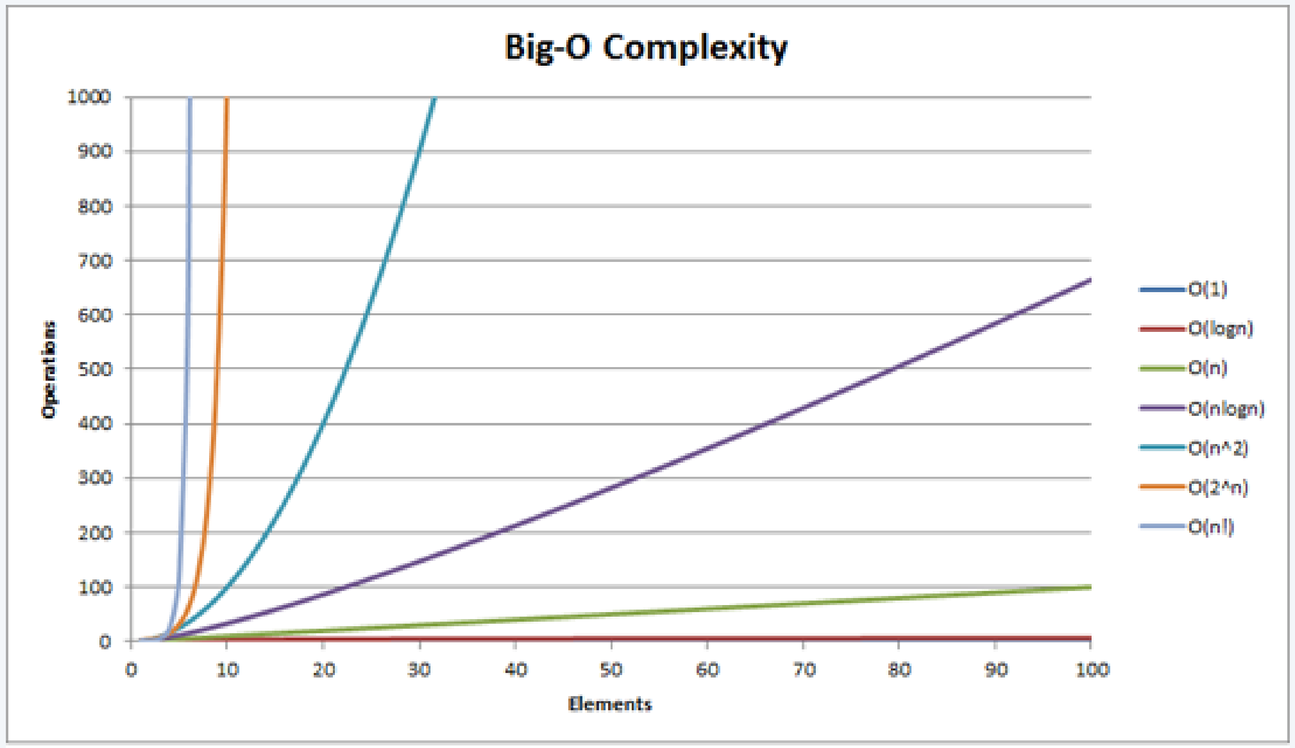
A. ~ $N^3/2$ nanoseconds.

✓ ⟵ matches $4.84 \times 10^{-10} \times N^3$ empirical hypothesis

Comparing algorithms

Recognize typical orders of complexity (O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), O(2^n))

LO 11.1b



Q. Order algorithms from slowest to fastest

A. $n! > 2^n > n^n > n \log n > n > \log n > 1$

Classify Algorithms

Categorize algorithms according to their Big O complexity.

LO 11.1e

Q. Classify each of these algorithms using typical orders of complexity.

Algorithm. *Find an element in a random array of N ints* ⟵ N

Algorithm. *Swap any two elements in an array of size N* ⟵ 1

Algorithm. *Given N integers, find 3 integers that sums to zero* ⟵ N^3

Algorithm. *Find an element in a sorted array of N ints* ⟵ Log N

Algorithm. *Given N elements, find all combinations* ⟵ N!

Counting basic operations

Identify the basic operations in an algorithm and determine the running time of an algorithm by counting its basic operations.

LO 11.1c

```
public static int count(int[] a)
{
   int N = a.length;                    ←—— 1
   int cnt = 0;                         ←—— 1
   for (int i = 0; i < N; i++)          ←—— 1 initialization, N+1 comparisons, N increments
         if (a[i] < 0)                  ←—— N comparisons
           cnt++;                       ←—— 0 to N increments
   return cnt;
}
```

Q. What types of basic operations are performed in the code above?

A. *Assignment (=) , comparison (<), increment (++)*

Counting operations

Express the running time of an algorithm as a function of the input size and derive the complexity class of the algorithm.

LO 11.1d

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 0;
   for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
            if (a[i] + a[j] == 0)
              cnt++;
   return cnt;
}
```

Q. What types of basic operations are performed in the code above?

A. *Assignment (=) , comparison (<, ==), addition (+) increment (++)*

Counting comparisons (==) operations (if statement)

i	j	operations
0	1….N-1	N-1
1	2…N-1	N-2
2	3…N-1	N-3
.		
.		
N-2	N-1..N-1	1
N-1	N..N-1	0

Total Comparisons = (N-1) + (N-2) + … + 2 + 1

~ N^2

Running time is proportional to: N^2

Complexity class of the algorithm: N^2

Common order-of-growth hypothesis

Categorize algorithms according to their Big O complexity.

LO 11.1e

description	*order of growth*	*example*	*framework*
constant	1	`count++;`	*statement (increment an integer)*
logarithmic	$\log n$	`for (int i = n; i > 0; i /= 2)` `  count++;`	*divide in half (bits in binary representation)*

linear	n	`for (int i = 0; i < n; i++)`
		`  if (a[i] == 0)`
		`  count++;`
			single loop (check each element)

linearithmic	$n \log n$	[*see mergesort* (PROGRAM 4.2.6)]	*divide-and-conquer (mergesort)*

quadratic	n^2	`for (int i = 0; i < n; i++)`
		`  for (int j = i+1; j < n; j++)`
		`  if (a[i] + a[j] == 0)`
		`  count++;`
			double nested loop (check all pairs)

cubic	n^3	`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++;`
			triple nested loop (check all triples)

exponential	2^n	[*see Gray code* (PROGRAM 2.3.3)]	*exhaustive search (check all subsets)*

Patterns. Look for specific framework to determine order-of-growth.

Time and space efficiency

Describe space and time efficiency tradeoffs when designing algorithms.

LO 11.1f

Time efficiency. How fast a program runs (on average)?

Space efficiency. How much memory is used by a program (on average)?

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

primitive types

← Typical memory used by primitive types

Finding best, average and worst cases

Identify the best case, worst case, and average case Big O complexities of algorithms.

LO 11.1g

Best case. Describe the performance of an algorithm under optimal conditions.

Worst case. A safe analysis of the performance of an algorithm under worst conditions.

Average case. A safe analysis of the performance of an algorithm under average conditions.

ALGORITHM	BEST	AVERAGE	WORST	REMARKS
selection sort	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges; quadratic in best case
insertion sort	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small or partially-sorted arrays
bubble sort	n	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	rarely useful; use insertion sort instead
mergesort	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable

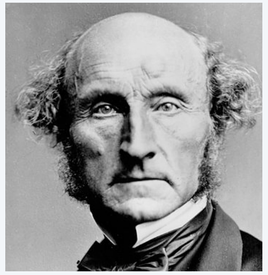
Context

Scientific method

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

Francis Bacon
1561–1626

René Descartes
1596–1650



John Stuart Mill
1806–1873

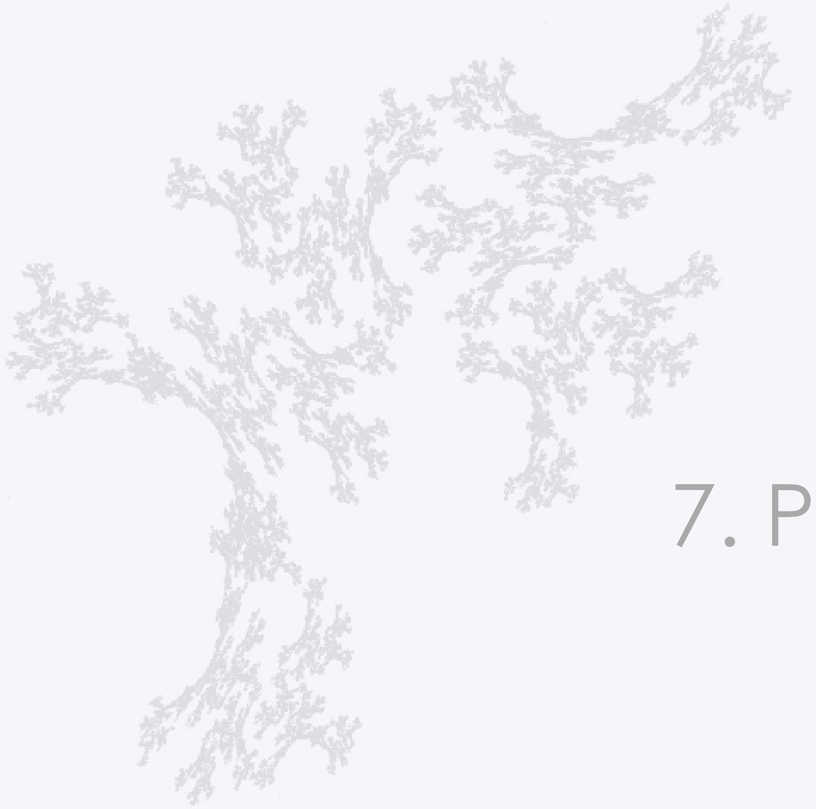
Empirical analysis of programs

- "Feature of natural world" is time taken by a program on a computer.
- Fit a curve to experimental data to get a formula for running time as a function of *N*.
- Useful for predicting, but not *explaining*.

Mathematical analysis of algorithms

- Analyze *algorithm* to develop a formula for running time as a function of *N*.
- Useful for predicting *and* explaining.
- Might involve advanced mathematics.
- Applies to any computer.

Good news. Mathematical models are easier to formulate in CS than in other sciences.



7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

Key questions and answers

Q. Is the running time of my program ~ $a\ N^b$ seconds?

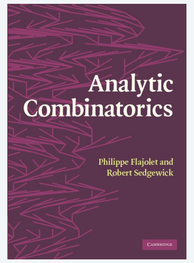
A. Yes, there's good chance of that. Might also have a $(\lg N)^c$ factor.

Q. How do you know?

A. Computer scientists have applied such models for decades to many, many specific algorithms and applications.

A. Programs are built from simple constructs (examples to follow).

A. Real-world data is also often simply structured.



A. Deep connections exist between such models and a wide variety of discrete structures (including some programs).

Doubling method

Hypothesis. The running time of my program is $T_N \sim a\, N^b$.

Consequence. As N increases, T_{2N}/T_N approaches 2^b.

no need to calculate a (!)

Proof: $\dfrac{a(2N)^b}{aN^b} = 2^b$

Doubling method

- Start with a moderate size.
- Measure and record running time.
- Double size.
- Repeat while you can afford it.
- Verify that *ratios* of running times approach 2^b.
- Predict by *extrapolation*:

  multiply by 2^b to estimate T_{2N} and repeat.

3-sum example

N	T_N	$T_N/T_{N/2}$
1000	0.5	
2000	4	8
4000	31	7.75
8000	248	8
16000	248 × 8 = 1984	8
32000	248 × 8^2 = 15872	8
…		
1024000	248 × 8^7 = 520093696	8

✓

math model says running time should be aN^3
$2^3 = 8$

Bottom line. It is often *easy* to meet the challenge of predicting performance.

Order of growth

Def. If a function $f(N) \sim a g(N)$ we say that $g(N)$ is the *order of growth* of the function.

Hypothesis. Order of growth is a property of the *algorithm*, not the computer or the system.

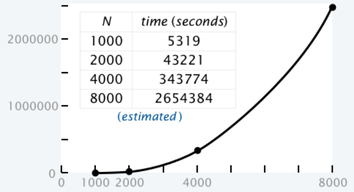
Experimental validation

Another hypothesis

1970s

VAX 11/780

$5.2 \times 10^{-6} \times N^3$ seconds



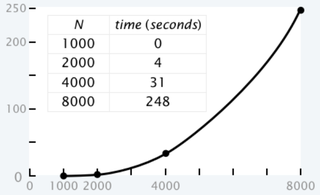
N	time (seconds)
1000	5319
2000	43221
4000	343774
8000	2654384

(estimated)

2010s: 10,000+ times faster

Macbook Air

$4.84 \times 10^{-10} \times N^3$ seconds



N	time (seconds)
1000	0
2000	4
4000	31
8000	248

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

When we execute a program on a computer that is X times faster, we expect the program to be X times faster.

Explanation with mathematical model

Mathematical model for 3-sum

```
public static int count(int[] a)
{
   int N = a.length;
   int cnt = 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)
               cnt++;
   return cnt;
}
```

operation	cost	frequency
function call/return	20 *ns*	1
variable declaration	2 *ns*	$\sim N$
assignment	1 *ns*	$\sim N$
less than compare	1/2 *ns*	$\sim N^3/6$
equal to compare	1/2 *ns*	$\sim N^3/6$
array access	1/2 *ns*	$\sim N^3/2$
increment	1/2 *ns*	$\sim N^3/6$

$$\#\ \texttt{i < j < k} = \binom{N}{3} = \frac{N(N-1)(N-2)}{6} \sim \frac{N^3}{6}$$

Q. Formula for total running time when return value is not large (typical case)?

A. $\sim N^3/2$ nanoseconds. ✓ ⟵ matches $4.84 \times 10^{-10} \times N^3$ empirical hypothesis

Machine- and system-dependent features of the model are all constants.

Order of growth

Recognize typical orders of complexity (O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), O(2^n))

LO 11.1b

Hypothesis. The order of growth of the running time of my program is $N^b (\log N)^c$.

log instead of lg since constant base not relevant

Evidence. Known to be true for many, many programs with simple and similar structure.

Linear (N)

```
for (int i = 0; i < N; i++)
  ...
```

Quadratic (N^2)

```
for (int i = 0; i < N; i++)
  for (int j = i+1; j < N; j++)
    ...
```

Cubic (N^3)

```
for (int i = 0; i < N; i++)
  for (int j = i+1; j < N; j++)
    for (int k = j+1; k < N; k++)
      ...
```

Logarithmic (log N)

```
public static void f(int N)
{
  if (N == 0) return;
  ... f(N/2)...
}
```

Linearithmic (N log N)

```
public static void f(int N)
{
  if (N == 0) return;
  ... f(N/2)...
  ... f(N/2)...
   for (int i = 0; i < N; i++)
      ...
 }
```

Exponential (2^N)

```
public static void f(int N)
{
  if (N == 0) return;
  ... f(N-1)...
  ... f(N-1)...
}
```

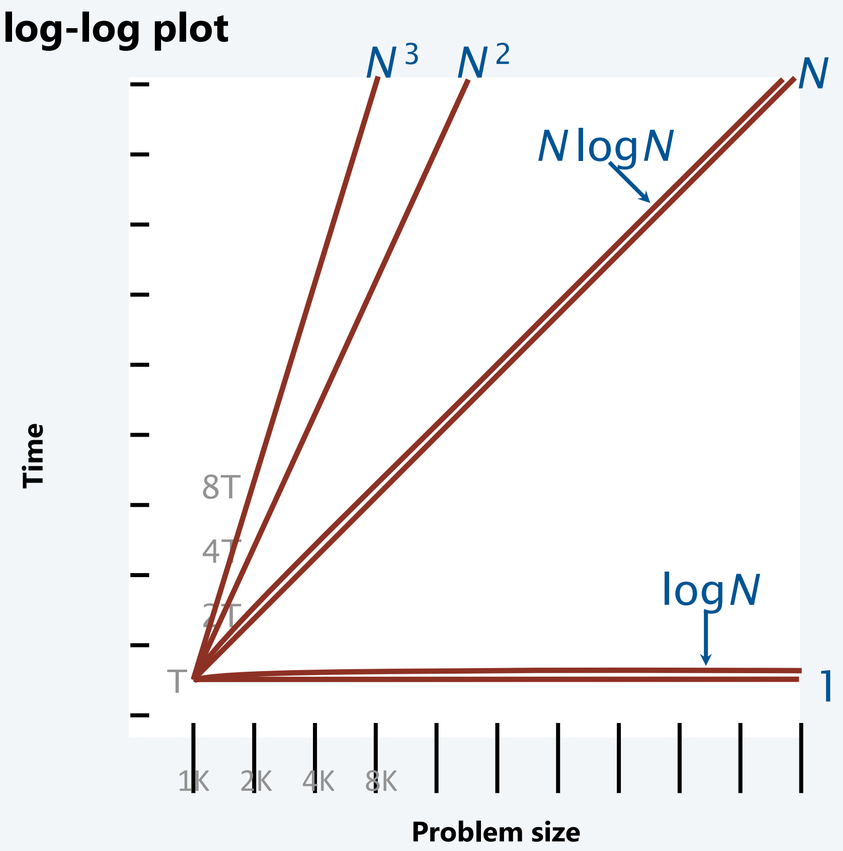
ignore for practical purposes (infeasible for large N)

Stay tuned for examples.

Order of growth classifications

order of growth		*slope of line in log-log plot* (b)	*factor for doubling method* (2^b)
description	*function*		
constant	1	0	1
logarithmic	log$N$	0	1
linear	N	1	2
linearithmic	N log$N$	1	2
quadratic	N^2	2	4
cubic	N^3	3	8

if input size doubles running time increases by this factor



math model may have log factor

If math model gives order of growth, use doubling method to validate 2^b ratio.

If not, use doubling method and solve for $b = \lg(T_N / T_{N/2})$ to estimate order of growth to be N^b.

An important implication

Moore's Law. Computer power increases by a roughly a factor of 2 every 2 years.

Q. My *problem size* also doubles every 2 years. How much do I need to spend to get my job done?

a very common situation: weather prediction, transaction processing, cryptography...

Do the math

$T_N = aN^3$ running time today

$T_{2N} = (a/2)(2N)^3$ running time in 2 years

$= 4aN^3$

$= 4T_N$

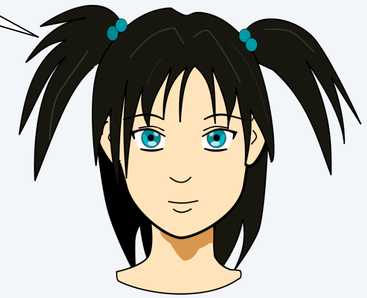
	now	*2 years from now*	*4 years from now*		*2M years from now*
N	$X	$X	$X	...	$X
$N \log N$	$X	$X	$X	...	$X
N^2	$X	$2X	$4X	...	$2^M X
N^3	$X	$4X	$16X	...	$4^M X

A. You can't afford to use a quadratic algorithm (or worse) to address increasing problem sizes.

Meeting the challenge

My program is taking too long to finish. I'm going out to get a pizza.

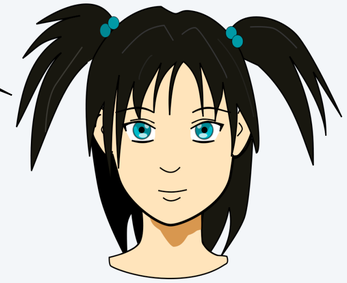
Mine, too. I'm going to run a doubling experiment.



Hmmm. Still didn't finish.

The experiment showed my program to have a higher order of growth than I expected. I found and fixed the bug.

Time for some pizza!

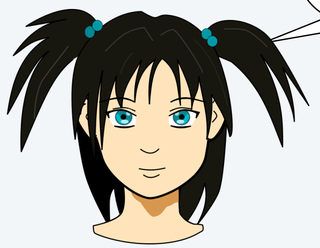


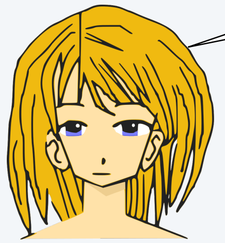
Doubling experiments provide good insight on program performance

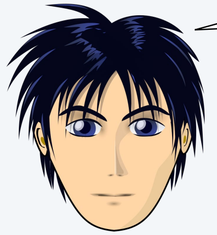
- Best practice to plan realistic experiments for debugging, anyway.
- Having *some* idea about performance is better than having *no* idea.
- *Performance matters* in many, many situations.

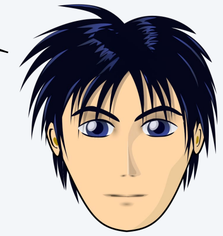
Caveats

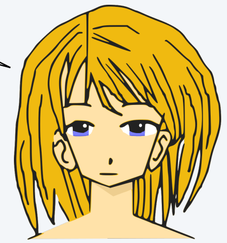
It is *sometimes* not so easy to meet the challenge of predicting performance.

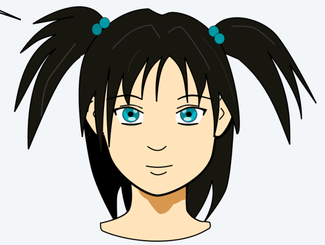






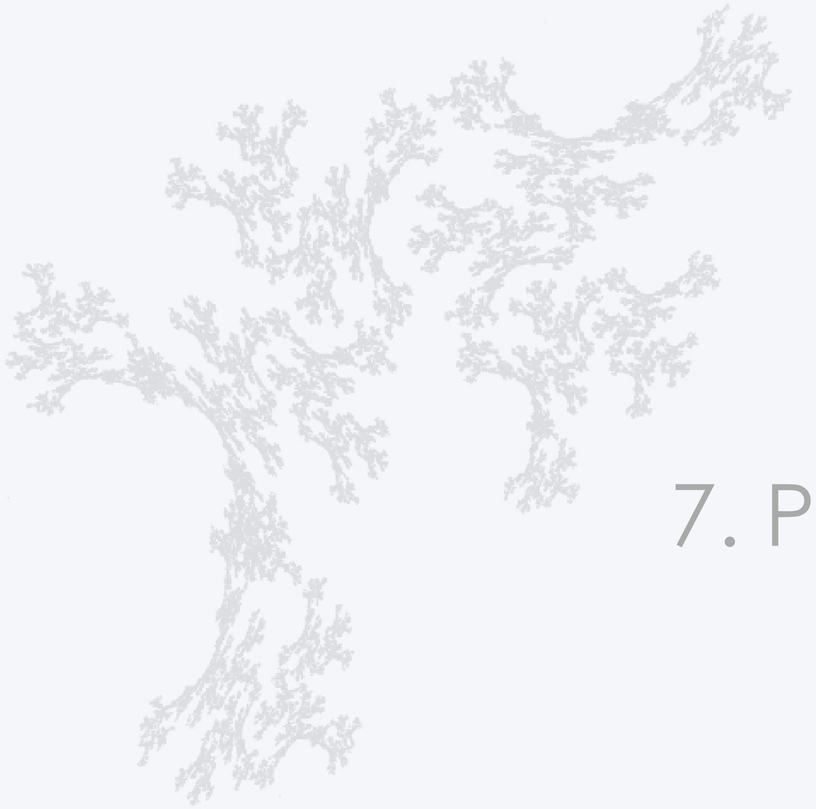






$$\frac{a(2N)^b(\lg(2N))^c}{aN^b(\lg N)^c} = 2^b\left(1+\frac{1}{(\lg N)}\right)^c \sim 2^b$$

Good news. Doubling method is *robust* in the face of many of these challenges.



7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling hypothesis
- Familiar examples

Example: Gambler's ruin simulation

Q. How long to compute chance of doubling 1 million dollars?

```
public class Gambler
{
   public static void main(String[] args)
   {
     int stake  = Integer.parseInt(args[0]);
     int goal   = Integer.parseInt(args[1]);
     int trials = Integer.parseInt(args[2]);
     double start = System.currentTimeMillis()/1000.0;
     int wins   = 0;
     for (int i = 0; i < trials; i++)
     {
        int t = stake;
        while (t > 0 && t < goal)
        {
           if (Math.random() < 0.5) t++;
           else                     t--;
        }
        if (t == goal) wins++;
     }
     double now = System.currentTimeMillis()/1000.0;
     StdOut.print(wins + " wins of " + trials);
     StdOut.printf(" (%.0f seconds)\n", now - start);
   }
}
```

A. 4.8 million seconds (about 2 months).

N	T_N	$T_N/T_{N/2}$
1000	4	
2000	17	4.25
4000	56	3.29
8000	286	5.10
16000	1172	4.09
32000	$1172 \times 4 = 4688$	4
...		
1024000	$1172 \times 4^6 = 4800512$	4

✓

math model says order of growth should be N^2

```
% java Gambler 1000 2000 100
53 wins of 100 (4 seconds)
% java Gambler 2000 4000 100
52 wins of 100 (17 seconds)
% java Gambler 4000 8000 100
55 wins of 100 (56 seconds)
% java Gambler 8000 16000 100
53 wins of 100 (286 seconds)
```

```
% java Gambler 16000 32000 100
48 wins of 100 (1172 seconds)
```

Pop quiz on performance

Q. Let T_N be the running time of program Mystery and consider these experiments:

```
public class Mystery
{
   public static void main(String[] args)
   {
      ...
      int N = Integer.parseInt(args[0]);
      ...
   }
}
```

N	T_N (in seconds)	$T_N/T_{N/2}$
1000	5	
2000	20	4
4000	80	4
8000	320	4

Q. Predict the running time for $N = 64{,}000$.

Q. Estimate the order of growth.

Pop quiz on performance

Q. Let T_N be the running time of program Mystery and consider these experiments.

```
public class Mystery
{
   public static void main(String[] args)
   {
      ...
      int N = Integer.parseInt(args[0]);
      ...
   }
}
```

N	T_N (in seconds)	$T_N/T_{N/2}$
1000	5	
2000	20	4
4000	80	4
8000	320	4
16000	320 × 4 = 1280	4
32000	1280 × 4 = 5120	4
64000	5120 × 4 = 20480	4

Q. Predict the running time for $N = 64{,}000$.

A. 20480 seconds.

Q. Estimate the order of growth.

A. N^2, since lg 4 = 2.

Another example: Coupon collector

Q. How long to simulate collecting 1 million coupons?

```
public class Collector
{
   public static void main(String[] args)
   {
            int N = Integer.parseInt(args[0]);
            int trials = Integer.parseInt(args[1]);
            int cardcnt = 0;
     double start = System.currentTimeMillis()/1000.0;
            for (int i = 0; i < trials; i++)
      {
         int valcnt = 0;
               boolean[] found = new boolean[N];
         while (valcnt < N)
         {
                  int val = (int) (StdRandom() * N);
           cardcnt++;
                  if (!found[val])
             { valcnt++; found[val] = true; }
         }
      }
      double now = System.currentTimeMillis()/1000.0;
      StdOut.printf("%d %.0f ", N, N*Math.log(N) + .57721*N);
      StdOut.print(cardcnt/trials);
      StdOut.printf(" (%.0f seconds)\n", now - start);
   }
}
```

N	T_N	$T_N/T_{N/2}$
125000	7	
250000	14	2
500000	31	2.21 ✓
1000000	31 × 2 = 63	2

math model says order of growth should be $N \log N$

```
% java Collector 125000 100
125000 1539160 1518646 (7 seconds)
% java Collector 250000 100
250000 3251607 3173727 (14 seconds)
% java Collector 500000 100
500000 6849787 6772679 (31 seconds)
```

```
% java Collector 1000000 100
1000000 14392721 14368813 (66 seconds)
```

✓

A. About 1 minute. ← might run out of memory trying for 1 billion

Analyzing typical memory requirements

A *bit* is 0 or 1 and the basic unit of memory.

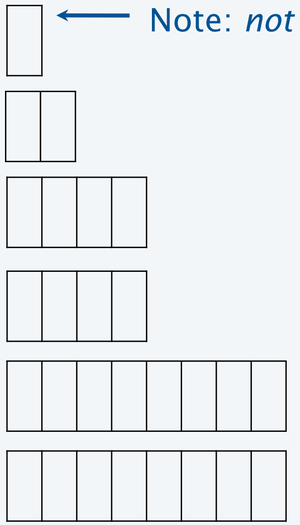
A *byte* is eight bits — the smallest addressable unit.

1 *megabyte* (MB) is about 1 million bytes.

1 *gigabyte* (GB) is about 1 billion bytes.

Primitive-type values

type	*bytes*
boolean	1
char	2
int	4
float	4
long	8
double	8



System-supported data structures (typical)

type	*bytes*
int[N]	4N + 16
double[N]	$8N + 16$
int[N][N]	$4N^2 + 20N + 16 \sim 4N^2$
double[N][N]	$8N^2 + 20N + 16 \sim 8N^2$
String	$2N + 40$

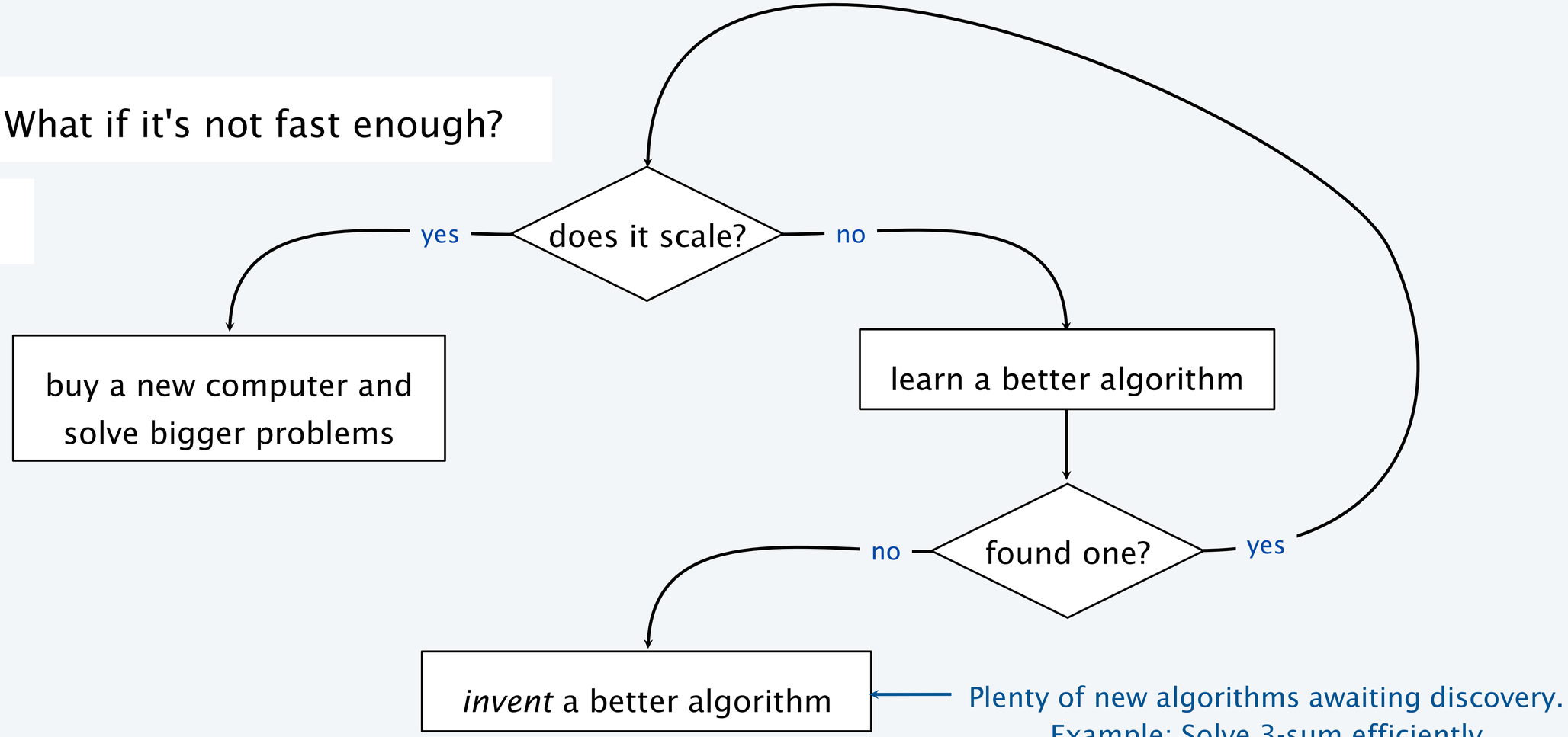
Example. 2000-by-2000 double array uses ~32MB.

Summary

Use computational experiments, mathematical analysis, and the *scientific method* to learn whether your program might be useful to solve a large problem.

Q. What if it's not fast enough?

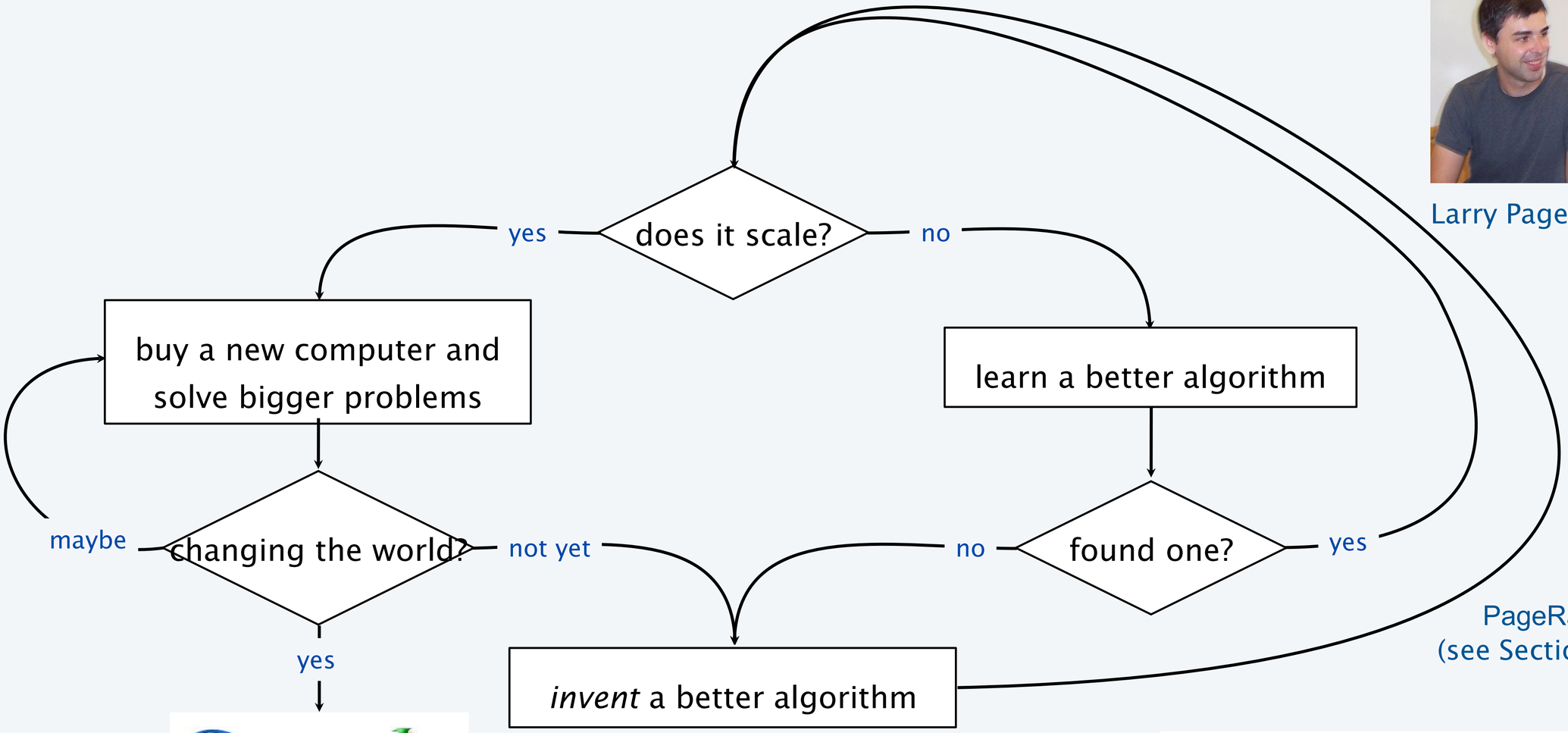
A.



Case in point

Not so long ago, 2 CS grad students had a program to index and rank the web (to enable search).

Larry Page and Sergey Brin



Google™

Lesson. Performance matters!

INTRODUCTION TO
COMPUTER SCIENCE

Rutgers University

Java ArrayList Class

Java ArrayList

As arrays an *ArrayList* also store large amounts of data in a single collection that can be referred to with a single variable.

The *ArrayList* class provides a dynamic array implementation

- stores objects of the same type sequentially
- grows as needed (double its size)
- useful if the size of the array is unknown
- implements the List interface

To declare an ArrayList

```
ArrayList <E> arrayReferenceVariable;
```

Item data type

Java ArrayList

An ArrayList implements the List interface

ArrayLists can only hold **objects**, and **not primitive** types such as ints.

- Each element in the sequence can be accessed separately.
- We can explicitly overwrite an object at a specified position in the sequence, thus changing its value.
- We can inspect the object at a specified location in the sequence.
- We can add an object into a specified position of the sequence.
- We can add an object to the end of the sequence.
- We can remove an object from a specified location in the sequence.

ArrayList operations

To declare and instantiate an ArrayList of Strings:

```
ArrayList<String> names = new ArrayList<String>();
```

int size()
returns the number of elements in this list

boolean add(x)
appends x to the end of list; returns true

void add(index, x)
inserts x at position index, sliding elements at position index and higher to the right. Adds 1 to their indices and adjusts size

E get(index)
returns the element at the specified position in this list.

E set(index, x)
replaces the element at index with x
returns the element formerly at the specified position

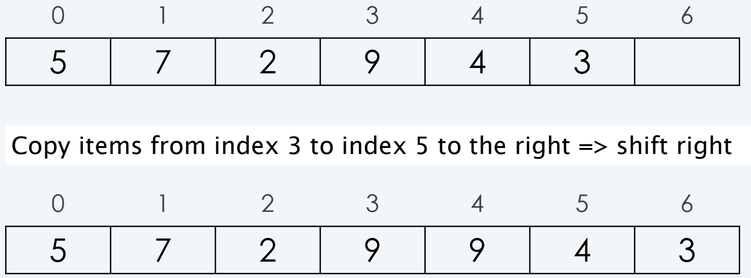
E remove(index)
removes element from position index, sliding subsequent elements to the left (subtracts 1 from their indices)
and adjusts size and returns the element at the specified position in this list.

E remove(obj)
removes element from position index, sliding subsequent elements to the left (subtracts 1 from their indices)
and adjusts size and returns the element at the specified position in this list.

int indexOf(obj)
If obj is found in the list, then the position number where it is found is returned.
If the obj is not found, then -1 is returned.

Array and ArrayList

- Arrays are a fixed length.
- ArrayLists grow and shrink as needed.
- Arrays can hold primitive values or objects.
- ArrayLists hold objects.



	Array	ArrayList	Best case	Worst case
Can access each element separately	`arr[i]`	`list.get(i);`	O(1)	O(1)
Can explicitly overwrite an element in a specific position	`arr[i] = "Tom";`	`list.set(i,"Tom");`	O(1)	O(1)
Can add an object into a specified position in the sequence	If there is space, copy items i…length-1 to the right, then insert object into position i	`list.add(i,"Joe");`	O(1)	O(n)
Can add an object to the end	If there is space, add object at the end (must keep track of last added element)	`list.add("Joe");`	O(1)	O(n)
Can remove an object from a specified location	Copy items (i+1)…(length-1) to the left. Remove last element.	`list.remove(i);`	O(1)	O(n)

Problem 1

Write a method, *numWordsOfLength*, that has two parameters, an *ArrayList<String> list*, and an integer *len*. The method returns the number of words in list that are exactly *len* letters long.

For example,
If list = ["cat", "dog", "mouse", "rat", "frog"]

numWordsOfLength(list, 3) returns 3;
numWordsOfLength(list, 4) returns 1;
numWordsOfLength(list, 2) returns 0;

Problem 2

Write a method, *removeWordsOfLength*, that has two parameters, an ArrayList *list*, and an integer *len*. The method removes all words in list that are exactly len letters long.

For example,
If list = ["cat", "dog", "mouse", "rat", "frog"]

removeWordsOfLength(list, 3) results with list = ["mouse", "frog"]
removeWordsOfLength(list, 4) results with list = ["cat", "dog", "mouse", "rat"]
removeWordsOfLength(list, 2) results with list = ["cat", "dog", "mouse", "rat", "frog"]

Image sources

https://openclipart.org/detail/25617/astrid-graeber-adult-by-anonymous-25617

https://openclipart.org/detail/169320/girl-head-by-jza

https://openclipart.org/detail/191873/manga-girl---true-svg--by-j4p4n-191873

Image sources

http://commons.wikimedia.org/wiki/File:Babbages_Analytical_Engine,_1834-1871._(9660574685).jpg

http://commons.wikimedia.org/wiki/File:Charles_Babbage_1860.jpg

http://commons.wikimedia.org/wiki/File:John_Tukey.jpg

http://commons.wikimedia.org/wiki/File:Andrew_Apple_(FloC_2006).jpg

http://commons.wikimedia.org/wiki/File:Hubble's_Wide_View_of_'Mystic_Mountain'_in_Infrared.jpg

Image sources

http://commons.wikimedia.org/wiki/File:KnuthAtOpenContentAlliance.jpg

http://commons.wikimedia.org/wiki/File:Pourbus_Francis_Bacon.jpg

http://commons.wikimedia.org/wiki/File:Frans_Hals_-_Portret_van_René_Descartes.jpg

http://commons.wikimedia.org/wiki/File:John_Stuart_Mill_by_London_Stereoscopic_Company,_c1870.jpg

Image sources

http://commons.wikimedia.org/wiki/File:FEMA_-_2720_-_Photograph_by_FEMA_News_Photo.jpg

http://pixabay.com/en/lab-research-chemistry-test-217041/

http://upload.wikimedia.org/wikipedia/commons/2/28/Cut_rat_2.jpg

http://pixabay.com/en/view-glass-future-crystal-ball-32381/

INTRODUCTION TO COMPUTER SCIENCE

Rutgers University

11. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples

Adopted and modified from Slides by Sedgewick and Wayne

INTRODUCTION TO
COMPUTER SCIENCE

Rutgers University

11. Performance

1.

http://introcs.cs.rutgers.edu