



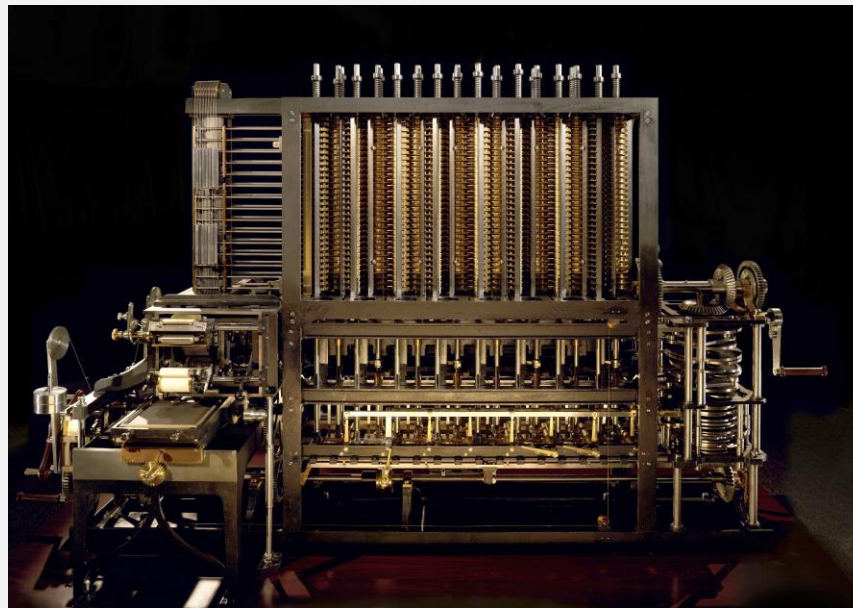
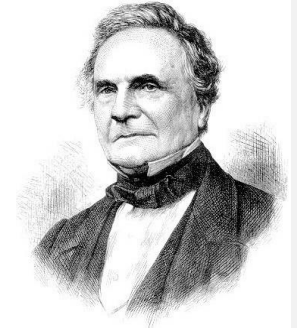
<http://algs4.cs.princeton.edu>

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth classifications*

Running time

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



how many times do
you have to turn
the crank?



Concerns about running time preceded actual working computers
by almost a century!

One of the early achievements of computer science

The ability to estimate and bound the running time of a piece of code
as a function of the size of the input
without seeing the actual input data
and only minimal knowledge of the system it will run on



1.4 ANALYSIS OF ALGORITHMS

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

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course
(REIR)

theory of algorithms
(HGRE)

Primary practical reason: avoid performance bugs.



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



Scientific method applied to the analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.

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



Principles.

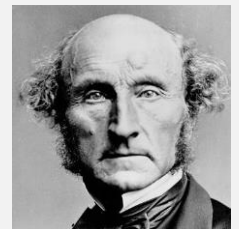
- Experiments must be **reproducible**.
- Hypotheses must be **falsifiable**.



Francis
Bacon



René
Descartes



John Stuart
Mills

Feature of the natural world. Computer itself.



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

1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth*
classifications

Example: 3-SUM

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

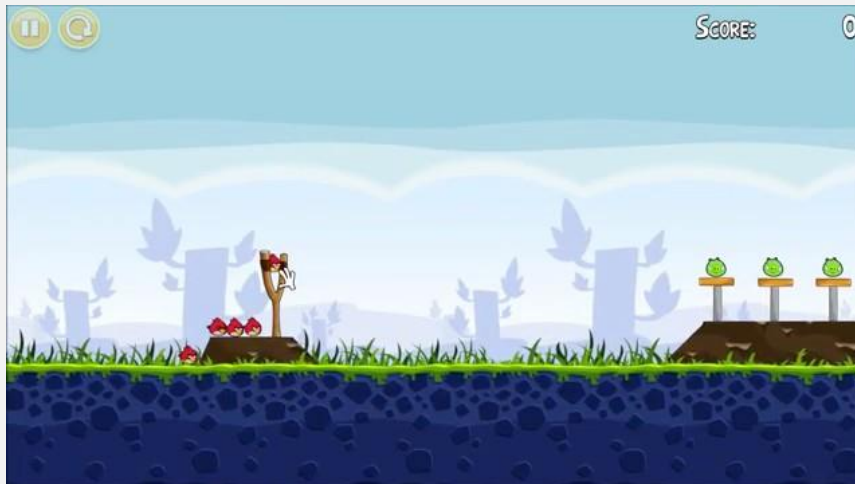
Input:

30 -40 -20 -10 40 0 10 5

Output:

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 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;
}
```

← check each triple

Ignore integer overflow in computing $a[i] + a[j] + a[k]$

A. Manual.

[illegible]

Measuring the running time

Q. How to time a program?

A. Automatic.

<code>public class Stopwatch</code>	<code>(part of algs4.jar)</code>
<code>Stopwatch()</code>	<i>create a new stopwatch</i>
<code>double elapsedTime()</code>	<i>time since creation (in seconds)</i>

```
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.

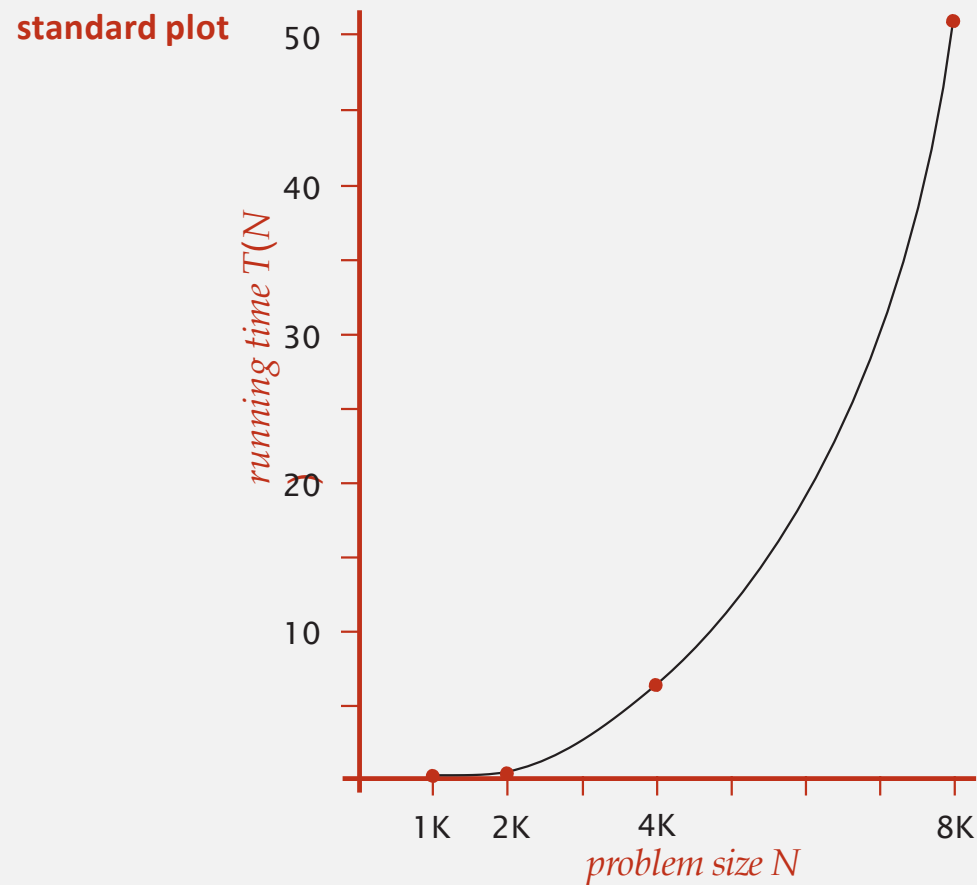
n	time (seconds) †
250	0
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

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

(not consistent with prev. slide)

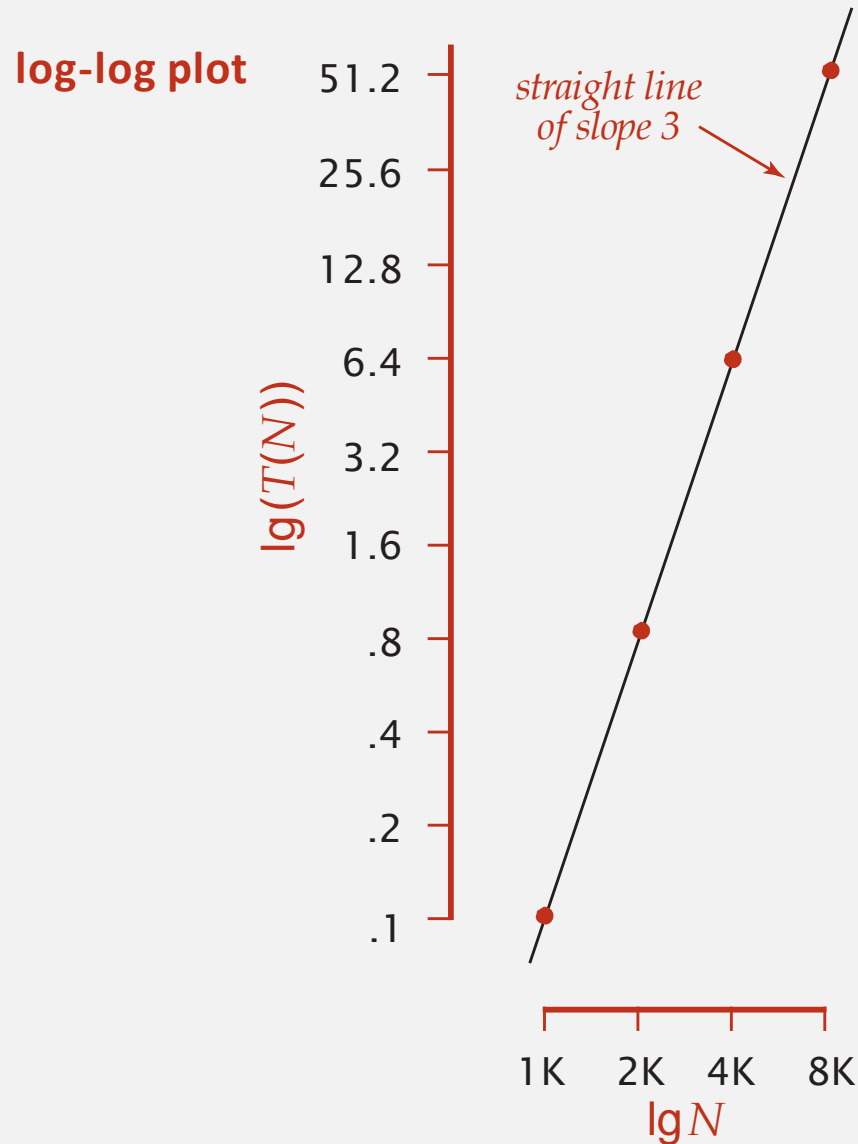
Data analysis

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**.



$$T(n) = a n^b$$

$$\lg(T(n)) = b \lg n + \lg(a)$$

Slope = b

y-intercept = $\lg(a)$

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

\lg = base 2 logarithm

Prediction and validation

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

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

Predictions.

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

Observations.

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

validates hypothesis!

Doubling hypothesis

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

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

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

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

← $\lg(6.4 / 0.8) = 3.0$

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

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

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

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

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

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

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

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

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

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



almost identical hypothesis to one
obtained via log-log plot

Analysis of algorithms quiz 1

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

- A.** 20 *seconds*.
- B.** 52 *seconds*.
- C.** 117 *seconds*.
- D.** 350 *seconds*.
- E.** *I don't know.*

n	time (seconds) †
1000	0.02
2000	0.05
4,000	0.2
8,000	0.81
16,000	3.25



1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth*
classifications

Variance

Variability. Different (random) runs of the same algorithm may require quite different time. This can affect the analysis.

N=8000					
3,4	3,7	6,2	2,2	4,1	2,3



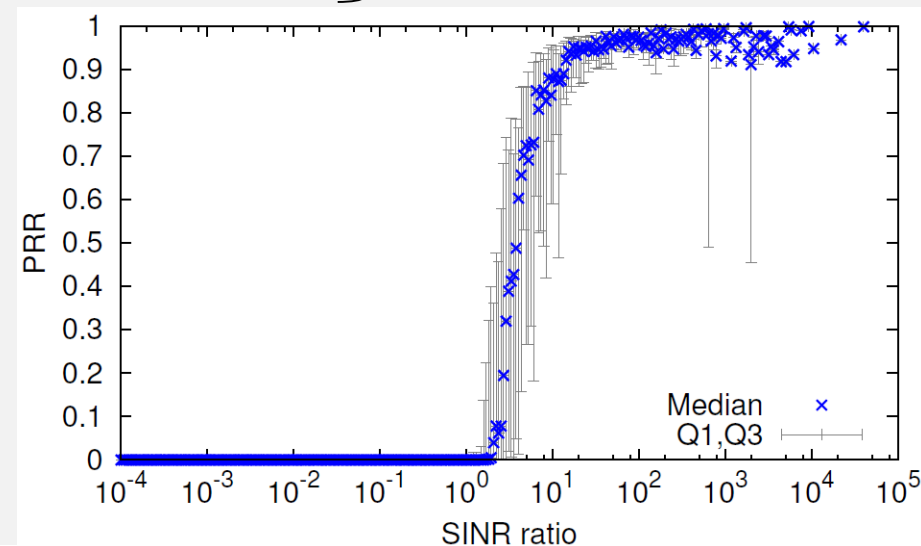
Standard solution.

- Perform multiple runs, and use their **average**. 3,65
- Alternatively, their **median**. 3,55

StdStats (algs4)

Variance. Computing the **standard deviation** tells us how much the measurements vary.

Alternatively, **confidence intervals** express even better how the measurements vary.



Two surprises

Approximate running time is a simple mathematical expression

Generally holds true even for much more complex programs!

Running time on different systems differs only by a constant factor!

Running time on system 1: $a_1 n^b$

Running time on system 2: $a_2 n^b$

Experimental algorithmics

System independent effects.

- Algorithm.
 - (Rarely) Input data.
- } determines exponent b
in power law $a n^b$

System dependent effects.

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

} determines constant a in
power law $a n^b$

Theorist vs. pragmatist view of algorithmic efficiency

$$a n^b$$

↑ system-dependent

← property of algorithm

Theorist: Worrying about constant factors is tedious and crass!
The asymptotic efficiency of an algorithm is a mathematical fact.
I study properties of the universe. The computer is irrelevant!

Novice: My program ran in 3 seconds on my laptop when I fed it
data. That's pretty good, right?

Pragmatist: I will use math. model to compute b , then verify empirically.
I will use a combination of math and observation to estimate a .

Bad news. Sometimes difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

An aside

Algorithmic experiments are virtually free by comparison with other sciences.



Chemistry
(1 experiment)



Physics
(1 experiment)



Computer Science
(1 million experiments)

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



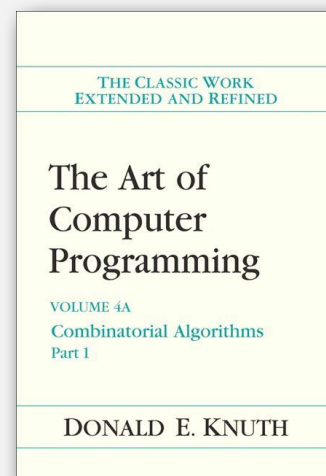
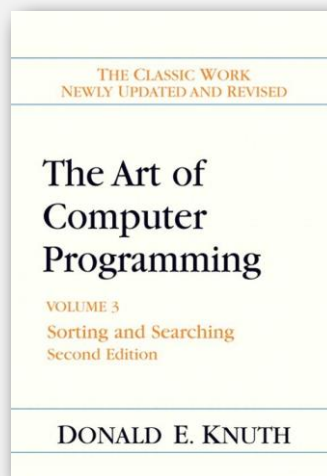
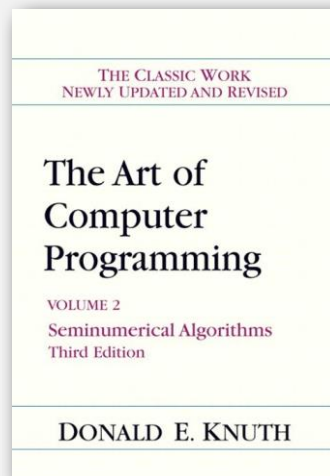
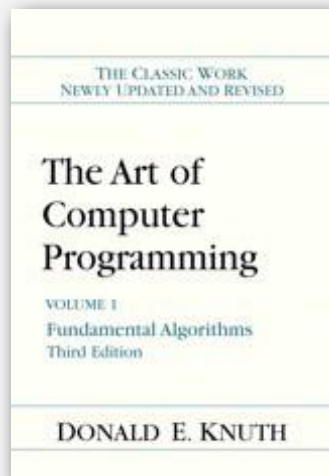
1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth*
classifications

Mathematical models for running time

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

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



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

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	<code>Math.sin(theta)</code>	91.3
arctangent	<code>Math.atan2(y, x)</code>	129
	...	

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

Frequency of basic operations: 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

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++)
    if (a[i] == 0)
        count++;
```

Heuristic: pick an operation that's both frequent and costly

Assumption:

Array access dominates running time

This is a hypothesis that can be tested

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

Memory access is a good candidate:
communicating outside CPU is often costly

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

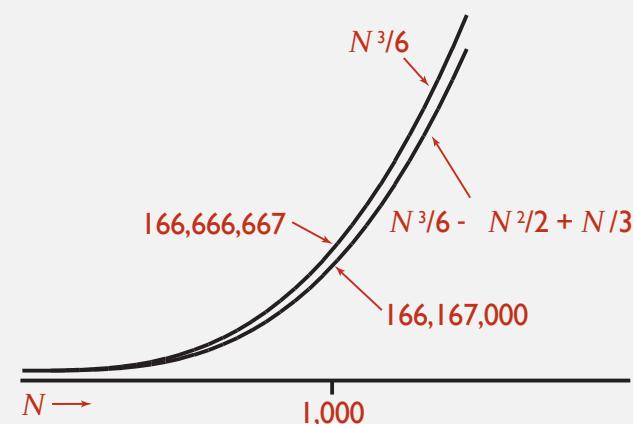
Ex 1. $\frac{1}{6} n^3 + 20n + 16 \sim \frac{1}{6} n^3$

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

Ex 3. $\frac{1}{6} n^3 - \underbrace{\frac{1}{2} n^2 + \frac{1}{3} n}_3 \sim \frac{1}{6} n^3$

discard lower-order terms

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



Leading-term approximation

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

Analysis of algorithms quiz 2

What is the tilde simplification of the following expression?

$$\frac{3}{2} n^2 \lg n + 100 n + n^3$$

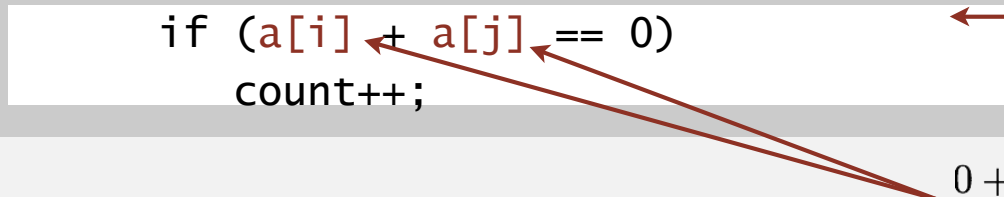
- A. $\sim n^2 \lg n$
- B. $\sim \frac{3}{2} n^2 \lg n$
- C. $\sim 100 n$
- D. $\sim n^3$
- E. *I don't know.*

Example: 2-SUM

Q. Approximately how many **array accesses** as a function of input size n ?

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

"inner loop"


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

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

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

Example : 3-SUM

Q. Approximately how many **array accesses** as a function of input size n ?

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

"inner loop"

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

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


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

Analysis of algorithms quiz 3

How many array accesses does the following code fragment make as a function of n ?

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

- A. $\sim n^2 \lg n$
- B. $\sim \frac{3}{2} n^2 \lg n$
- C. $\sim \frac{1}{2} n^3$
- D. $\sim \frac{3}{2} n^3$
- E. *I don't know.*

$k = 1, 2, 4, \dots$

 $\lg n$ times



1.4 ANALYSIS OF ALGORITHMS

- *introduction*
- *observations*
- *mathematical models*
- *order-of-growth*
classifications

Common order-of-growth classifications


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

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

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

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

Typical usage. Mathematical analysis of running times.

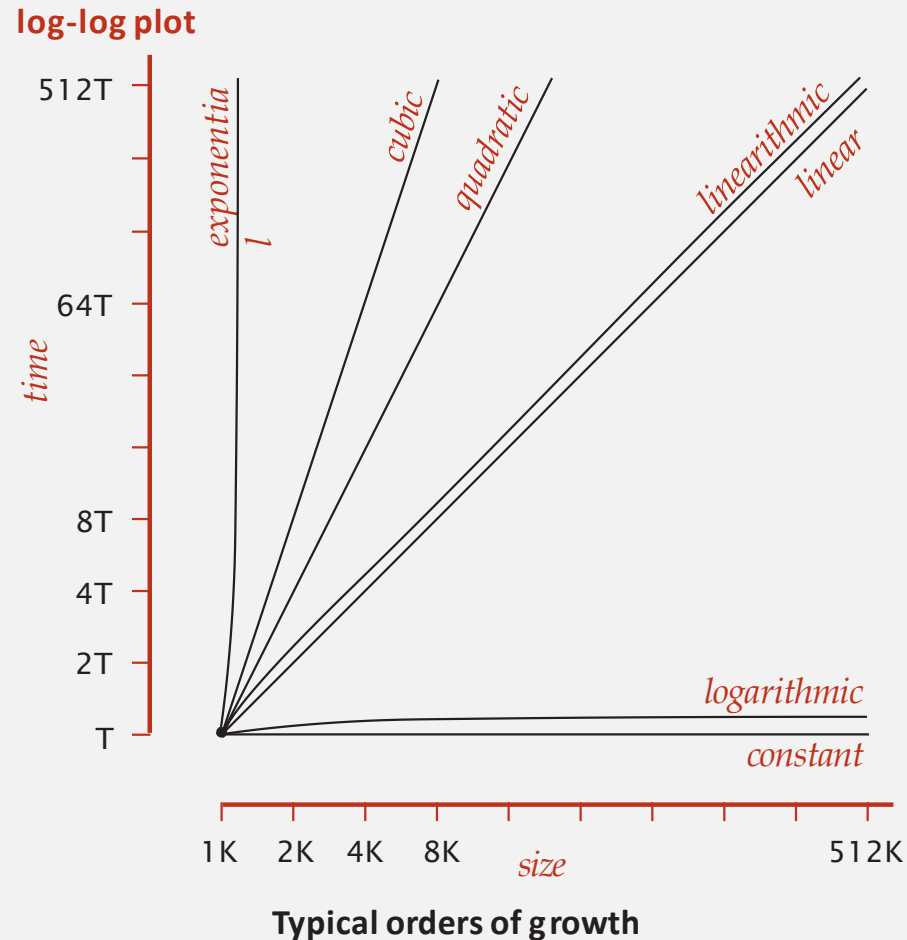
 where leading coefficient
depends on machine, compiler, JVM, ...

Common order-of-growth classifications

Good news. The set of functions

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

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



Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2n) / T(n)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log n$	logarithmic	<code>while (n > 1) { n = n/2; ... }</code>	divide in half	binary search	~ 1
n	linear	<code>for (int i = 0; i < n; i++) { ... }</code>	single loop	find the maximum	2
$n \log n$	linearithmic	<i>see mergesort lecture</i>	divide and conquer	mergesort	~ 2
n^2	quadratic	<code>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) { ... }</code>	double loop	check all pairs	4
n^3	cubic	<code>for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) for (int k = 0; k < n; k++) { ... }</code>	triple loop	check all triples	8
	exponential	<i>see combinatorial search lecture</i>	exhaustive search	check all subsets	

The 3-sum problem: An $n^2 \log n$ algorithm

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
(or $n \log n$ with mergesort).
- 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

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

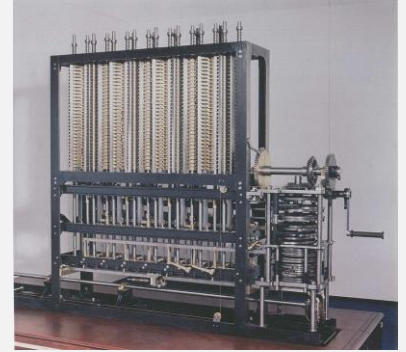
ThreeSumDeluxe.java

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

Turning the crank: summary

Empirical analysis.

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

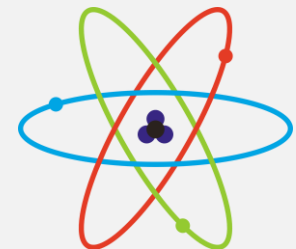


Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to **explain behavior**.

Scientific method.

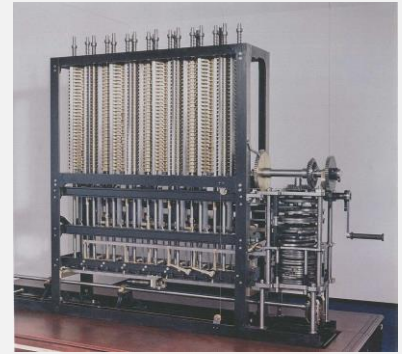
- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.



Summary

Empirical analysis.

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



Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to **explain behavior**.

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.

