# Announcements

## First programming assignment.

- Due Tomorrow at 11:00pm.
- Try electronic submission system today.
- "Check All Submitted Files." will perform checks on your code.
  - You may use this up to 10 times.
  - Can still submit after you use up your checks.
  - Should not be your primary testing technique!

## Registration.

- Register for Piazza.
- Register for Coursera.
- Register for Poll Everywhere.


http://algs4.cs.princeton.edu

# 1.4 Analysis of Algorithms

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


http://algs4.cs.princeton.edu

# 1.4 Analysis of Algorithms

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

# Efficiency

## 126 vs. 226

- 126: Techniques for solving problems.
- 226: Techniques for solving problems **efficiently.**

## Simple Example: Checking symmetry of an NxN matrix

- Naive: Scan all elements.
- Better: Scan only elements above the diagonal (>2x speedup).

# Efficiency (more insidious example)

```
public static String concatenateNoSpace(String s1, String s2) {
    for (int i = 0; i < s2.length(); i++)
        if (s2.charAt(i) != ‘ ‘)
            s1 = s1 + s2.charAt(i);
    return s1;
}
```

```
$ java-226 concatenateNoSpace
s2.length    Time (s)
  10000       0.14
  20000       0.41
  40000       1.62
  80000       6.59
```

**Common Problem.**

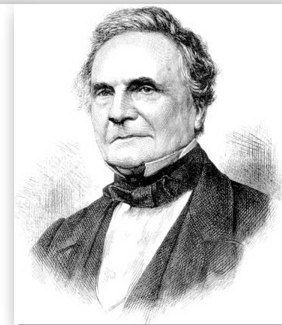
- Novice programmer does not understand performance characteristics of data structure.
- Results in poor performance that gets WORSE with input size.

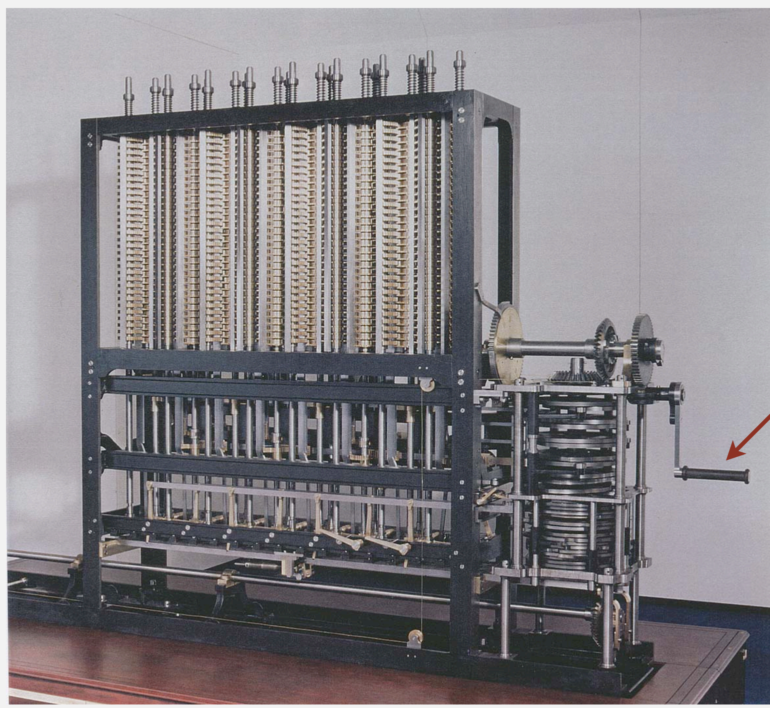
**Today**

- Precise definitions of program performance.
- Experimental and theoretical techniques for measuring performance.

# 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)*



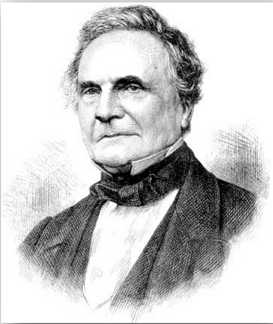


how many times do you have to turn the crank?

**Analytic Engine**

# The Life of the Philosopher

*" The iron folding-doors of the small-room or oven were opened. Captain Kater and myself entered, and they were closed upon us... The thermometer marked, if I recollect rightly, 265 degrees. The pulse was quickened, and I ought to have to have counted but did not count the number of inspirations per minute. Perspiration commenced immediately and was very copious. We remained, I believe, about five or six minutes without very great discomfort, and I experienced no subsequent inconvenience from the result of the experiment "* — *Charles Babbage, "From the Life of the Philosopher"*



265 Fahrenheit / 130 Celsius

# Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course

theory of algorithms

Primary practical reasons: avoid performance bugs

enable new technologies

example:simulation of galaxy formation

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

# Running time of programs





## Programs

- Mathematical objects.
- Running on physical hardware.

## Mathematical model

- Left program runtime: $cN^2$ Right program runtime: $c(N^2/2 - N/2)$

## Empirical observations

- Runtime of a program varies even when run on the same input.

# 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

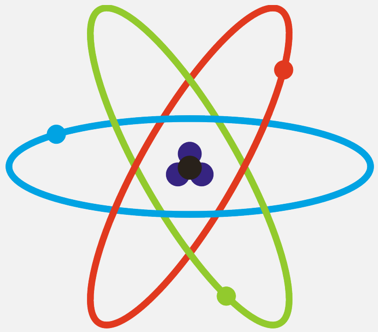
A framework for predicting performance and comparing algorithms.

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.

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

# 1.4 Analysis of Algorithms

- *introduction*
- *empirical 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)
   {
      int[] a = In.readInts(args[0]);
      StdOut.println(count(a));
   }
}
```

check each triple

for simplicity, ignore integer overflow

# Measuring the running time

Q. How to time a program?

A. Manual.

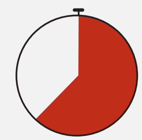
```
% java ThreeSum 1Kints.txt
```

*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)
{
   int[] a = In.readInts(args[0]);
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
}
```

# Empirical analysis

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

```
%
```

# Empirical analysis

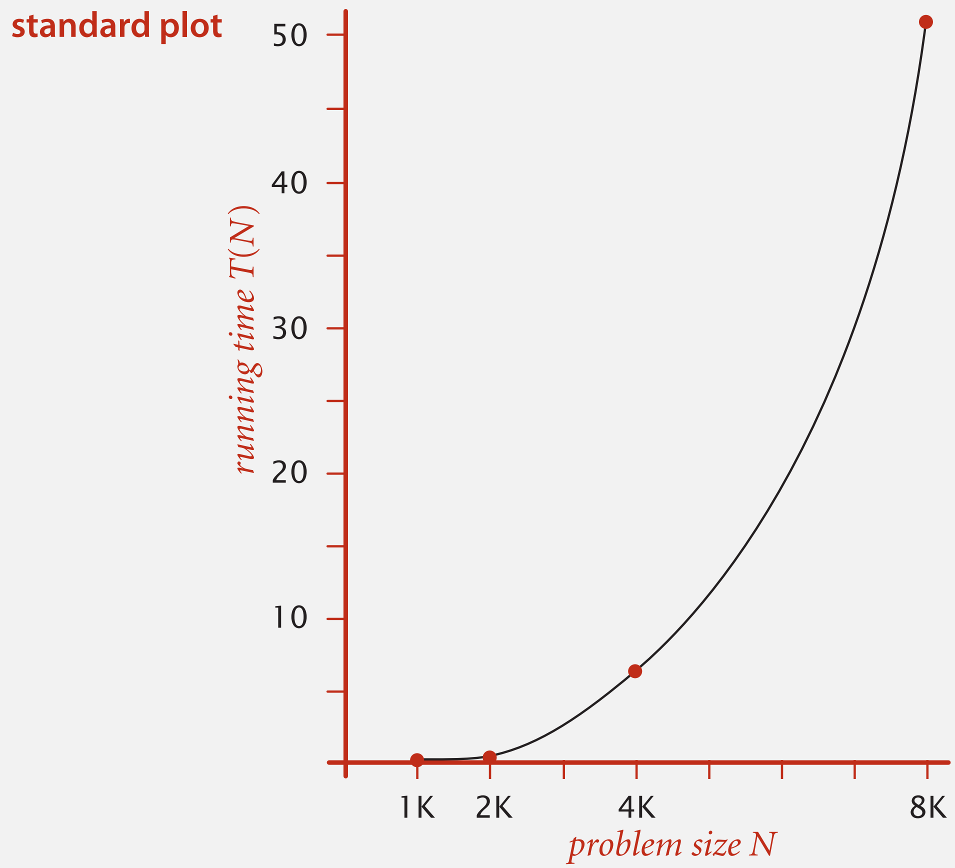
Run the program for various input sizes and measure running time.

| N | time (seconds) † |
|---|---|
| 250 | 0.0 |
| 500 | 0.0 |
| 1,000 | 0.1 |
| 2,000 | 0.8 |
| 4,000 | 6.4 |
| 8,000 | 51.1 |
| 16,000 | ? |

# Data analysis

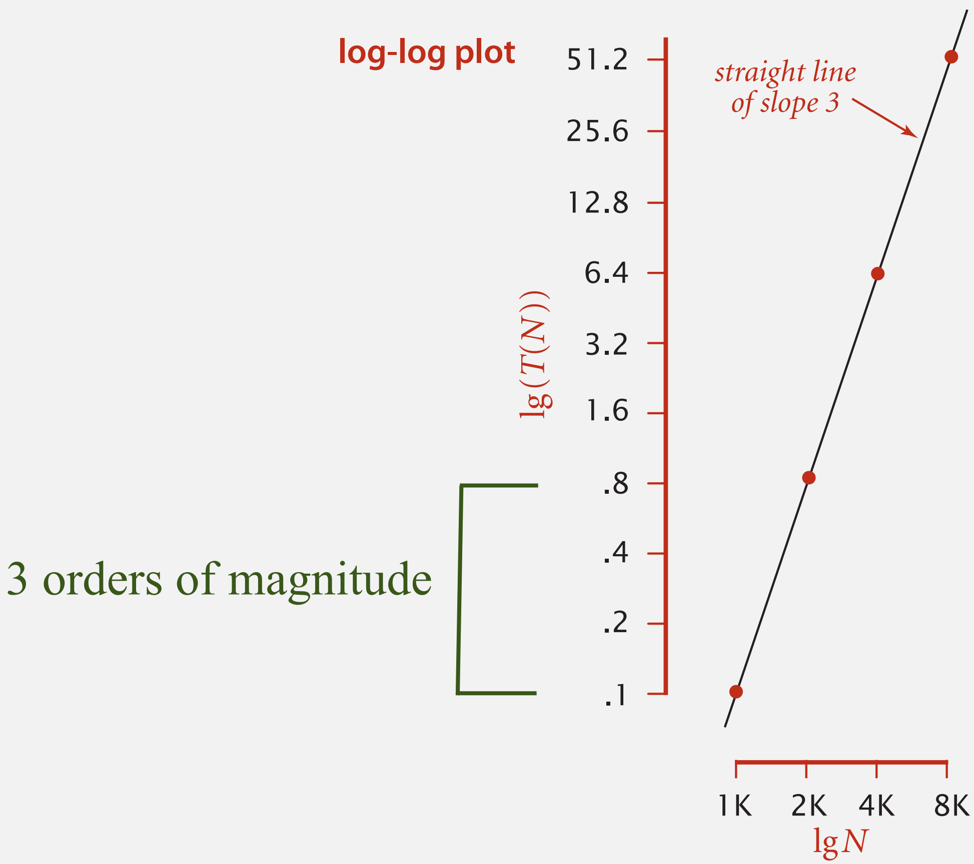
Standard plot. Plot running time $T(N)$ vs. input size $N$.

- Hard to form a useful hypothesis.



# Data analysis

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




$$T(N) = a\,N^{b}$$ ← power law

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

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

$$b = 2.999$$

$$c = -33.2103 \qquad a = 1.006 \times 10^{-10}$$

**Regression.** Fit straight line through data points: $\lg(T(N)) = b \lg(N) + c$

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

slope

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

# Prediction and validation

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

"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 | 51.0 |
| 8,000 | 51.1 |
| 16,000 | 410.8 |

validates hypothesis!

# Doubling hypothesis

**Doubling hypothesis.**

- Another way to build models of the form $T(N) = a N^b$
- Run program, doubling the size of the input.

| N | time (seconds) † | ratio | lg ratio |
|---|---|---|---|
| 250 | 0.0 | | – |
| 500 | 0.0 | 4.8 | 2.3 |
| 1,000 | 0.1 | 6.9 | 2.8 |
| 2,000 | 0.8 | 7.7 | 2.9 |
| 4,000 | 6.4 | 8.0 | 3.0 |
| 8,000 | 51.1 | 8.0 | 3.0 |

lg (51.122 / 6.401) = 3.0

seems to converge to a constant b ≈ 3

**Hypothesis.** Running time is about $a N^b$ with $b$ = lg ratio.

# 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 | 51.0 |
| 8,000 | 51.1 |

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

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

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 b in power law

System dependent effects.

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

determines constant a in power law

Caveat.

- In some cases, b can depend on system (e.g. virtualization)

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

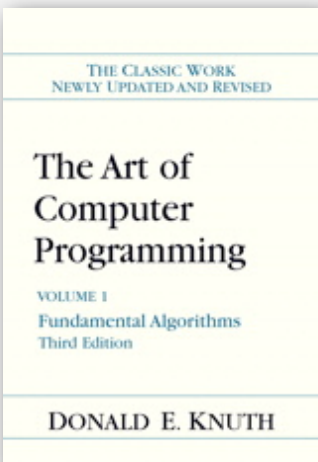
# 1.4 Analysis of Algorithms

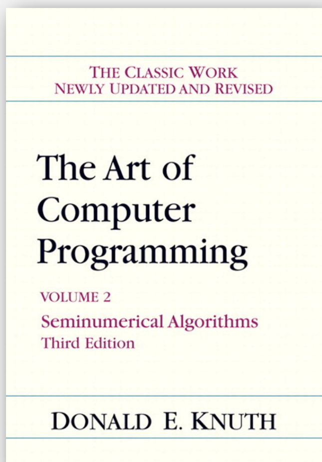
- *introduction*
- *empirical observations*
- ***mathematical models***
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

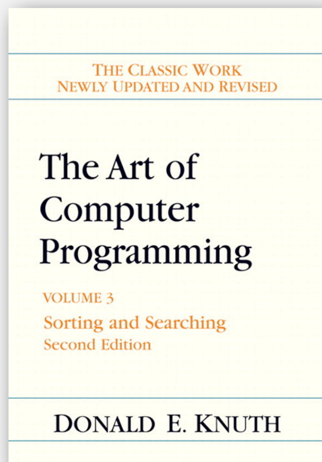
# Mathematical models for running time

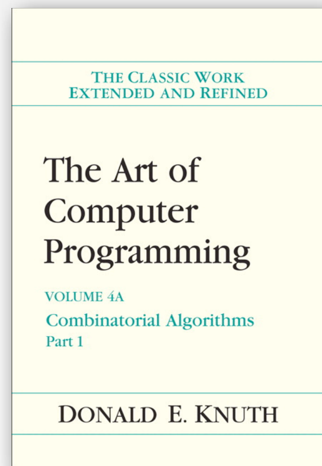
Total running time: sum of cost × frequency for all operations.

- 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.

# Timing basic operations (a hopeless endeavor)

| 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)` | 129.0 |
| ... | ... | ... |

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

Computer Architecture Caveats (see COS 475).

- Most computers are more like assembly lines than oracles (pipelining).
- Register vs. cache vs. RAM vs. hard disk (Java is a high level language)

# Cost of basic operations

| 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]` | $c_7 N^2$ |
| string length | `s.length()` | $c_8$ |
| substring extraction | `s.substring(N/2, N)` | $c_9$ |
| string concatenation | `s + t` | $c_{10} N$ |

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 | Frequency, N=10000 |
|---|---|---|
| variable declaration | 2 | 2 |
| assignment statement | 2 | 2 |
| less than compare | N + 1 | 10001 |
| equal to compare | N | 10000 |
| array access | N | 10000 |
| increment | N to 2 N | 10000 to 20000 |

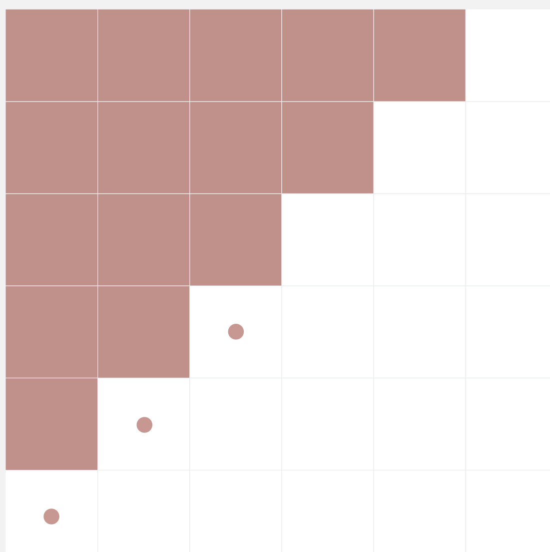
# Example: 2-SUM

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++;
```

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

Alternate Pf.



$$0 + 1 + 2 + \ldots + (N-1) = \underbrace{\frac{1}{2} N^2}_{\text{half of square}} - \underbrace{\frac{1}{2} N}_{\text{half of diagonal}}$$

# Example: 2-SUM

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

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

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

| 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++;
```

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

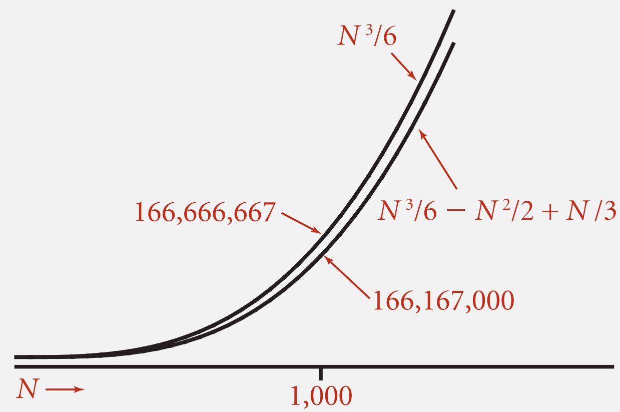
| operation | frequency |
|---|---|
| variable declaration | N + 2 |
| assignment statement | N + 2 |
| less than compare | ½ (N + 1) (N + 2) |
| equal to compare | ½ N (N − 1) |
| array access | N (N − 1) |
| increment | ½ N (N − 1) to N (N − 1) |

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

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

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

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

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

# Simplification 2: tilde notation

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

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

# Example: 2-SUM

Q. Approximately how many array accesses as a function of input size $N$?

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

"inner loop"

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

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

Because $2(½ N^2) = N^2$

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.

Because $(3/6\ N^3) = \frac{1}{2} N^3$

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

## Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take discrete mathematics course.

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

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

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

Ex 2. $1^k + 2^k + \ldots + N^k.$

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

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

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

Ex 4. 3-sum triple loop.

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

# Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take discrete mathematics course.

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

Ex 4. $1 + ½ + ¼ + ⅛ + \ldots$

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

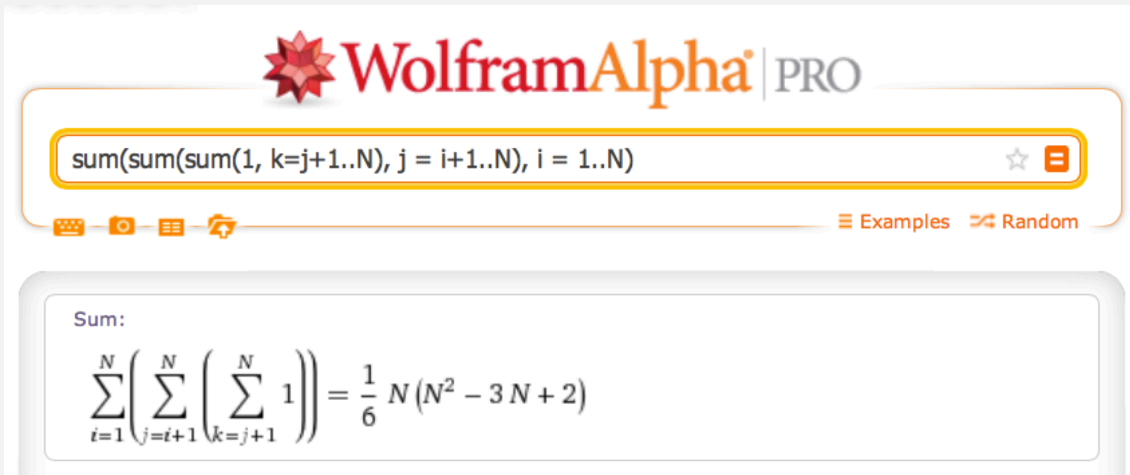
$$\int_{x=0}^{\infty} \left(\frac{1}{2}\right)^x dx = \frac{1}{\ln 2} \approx 1.4427$$

Caveat. Integral trick doesn't always work!

# Estimating a discrete sum

Q. How to estimate a discrete sum?

A3. Use Maple or Wolfram Alpha.




**wolframalpha.com**

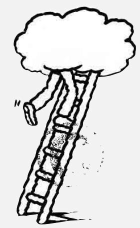
```
[wayne:nobel.princeton.edu] > maple15
    |\^/|     Maple 15 (X86 64 LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2011
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> factor(sum(sum(sum(1, k=j+1..N), j = i+1..N), i = 1..N));

                              N (N - 1) (N - 2)
                              -----------------
                                      6
```

# Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Realities of hardware impact accuracy of formulas.
- 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*
- *empirical observations*
- *mathematical models*
- *order-of-growth classifications*
- *theory of algorithms*
- *memory*

## Order-of-growth

Definition.

- If $f(N) \sim a\, g(N)$, then the order-of-growth of $f(N)$ is just $g(N)$
- Example:
  - Runtime of $3SUM: \sim 1/6\, t_1 N^3$ [see page 181]
  - Order-of-growth of the runtime of 3SUM: $N^3$
- We often say "order-of-growth of 3SUM" as shorthand for the runtime.

```
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++;
```

Time to execute: $t_1$

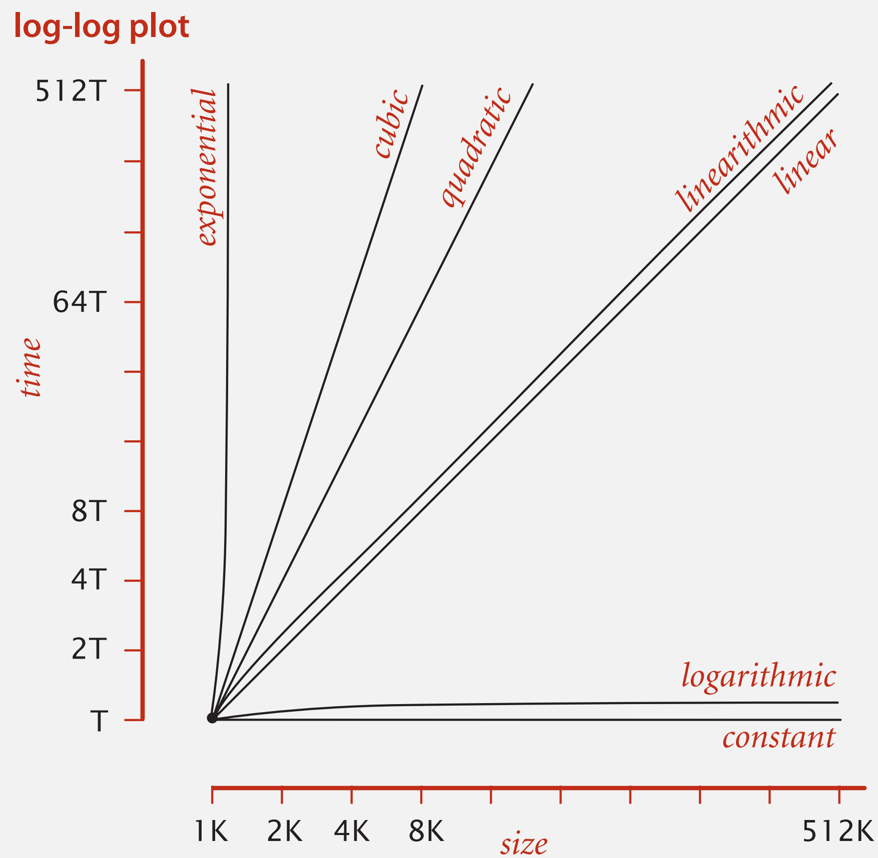
# Common order-of-growth classifications

Good news. the small set of functions

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

order of growth discards leading coefficient

suffices to describe order-of-growth of typical 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[0] + b[1];` | statement | add two array elements | 1 |
| log N | logarithmic | `while (N > 1)`<br>`{   N = N / 2;  ...   }` | divide in half | binary search | ~ 1 |
| N | linear | `for (int i = 0; i < N; i++)`<br>`{  ...        }` | 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++)`<br>` for (int j = 0; j < N; j++)`<br>`  {  ...        }` | double loop | check all pairs | 4 |
| $N^3$ | cubic | `for (int i = 0; i < N; i++)`<br>` for (int j = 0; j < N; j++)`<br>`  for (int k = 0; k < N; k++)`<br>`   {  ...        }` | triple loop | check all triples | 8 |
| $2^N$ | exponential | [see combinatorial search lecture] | exhaustive search | check all subsets | T(N) |

# Practical implications of order-of-growth

| growth rate | problem size solvable in minutes | | | |
|---|---|---|---|---|
| | 1970s | 1980s | 1990s | 2000s |
| 1 | any | any | any | any |
| log N | any | any | any | any |
| N | millions | tens of millions | hundreds of millions | billions |
| N log N | hundreds of thousands | millions | millions | hundreds of millions |
| $N^2$ | hundreds | thousand | thousands | tens of thousands |
| $N^3$ | hundred | hundreds | thousand | thousands |
| $2^N$ | 20 | 20s | 20s | 30 |

game changer

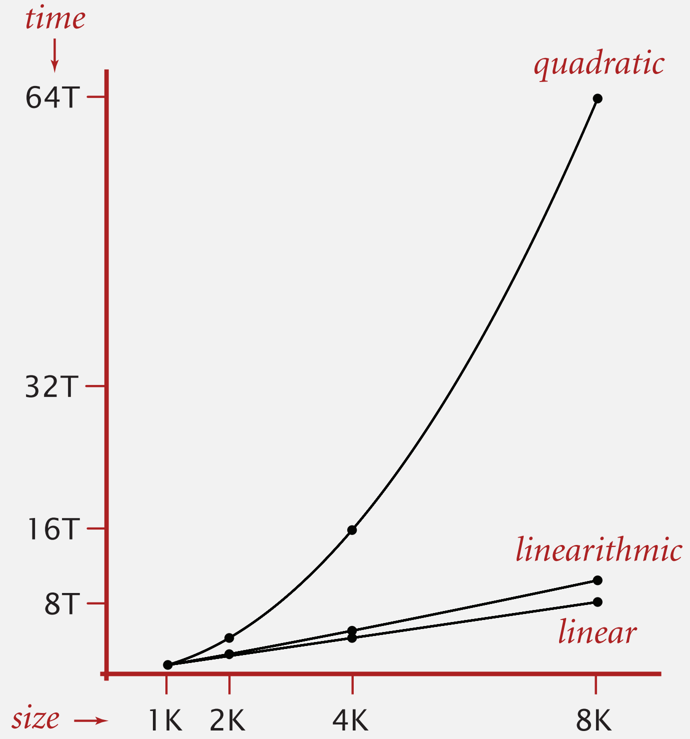
Bottom line. Need linear or linearithmic alg to keep pace with Moore's law.

# 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




c

# 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 |

Worst case: lg N

see Coursera for rigorous proof

# 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]`.

# An $N^2 \log N$ algorithm for 3-SUM

**Sorting-based 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.
  - $N^2$ binary searches, each $\log N$

**Remark.** Can achieve $N^2$ by modifying binary search step.

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 |
| ⋮ | ⋮ |
| (-40, 40) | ~~0~~ |
| ⋮ | ⋮ |
| (-20, -10) | 30 |
| ⋮ | ⋮ |
| (-10, 0) | 10 |
| ⋮ | ⋮ |
| ( 10, 30) | ~~-40~~ |
| ( 10, 40) | -50 |
| ( 30, 40) | -70 |

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

## Comparing programs

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

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

ThreeSum.java

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

ThreeSumDeluxe.java

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


http://algs4.cs.princeton.edu

# 1.4 Analysis of Algorithms

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

# Types of analyses: Performance depends on input

**Best case.** Lower bound on cost.

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

**Worst case.** Upper bound on cost.

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

**Average case.** Expected cost for random input.

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

**Ex 1.** Compares for binary search.

| | |
|---|---|
| Best: | $1$ |
| Average: | $\lg N$ |
| Worst: | $\lg N$ |

**Ex 2.** Array accesses for brute-force 3-SUM.

| | |
|---|---|
| Best: | $N^3$ |
| Average: | $N^3$ |
| Worst: | $N^3$ |

# Types of analyses: Performance depends on input

Best case. Lower bound on cost.

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

Worst case. Upper bound on cost.

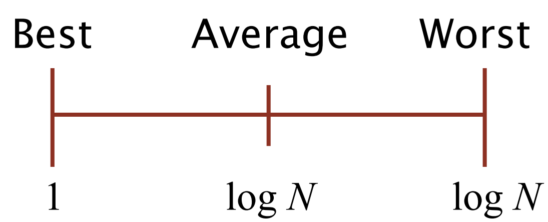
- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

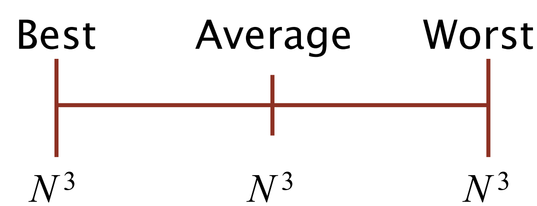
- Need a model for "random" input.
- Provides a way to predict performance.

Where you lie depends on your input!

Ex 1. Compares for binary search.




Ex 2. Array accesses for brute-force 3-SUM.



# Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost (guarantee).

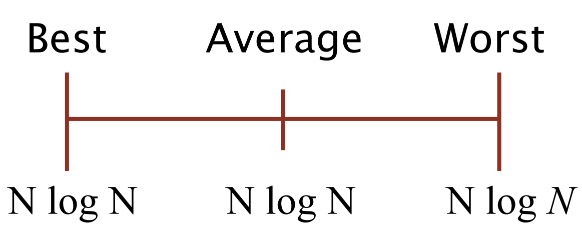
Average case. "Expected" cost.

Primary practical reason: avoid performance bugs.

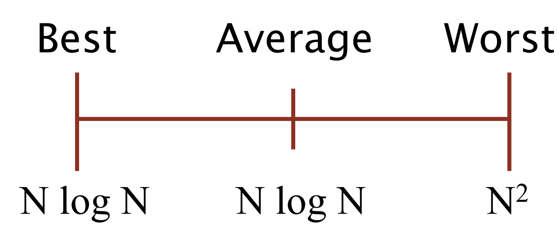
Example: Algorithm selection

- Given arbitrary data, performance may be anywhere in our bounds.
- Approach 1: depend on worst case guarantee.
  - Example: Use Mergesort instead of Quicksort
- Approach 2: randomize, depend on probabilistic guarantee.
  - Example: Randomize input before giving to Quicksort

Mergesort. [next week]




Quicksort. [next week]



# Theory of algorithms

## Previous slides

- Best, average, and **worst** case for a specific algorithm.

## New goals.

- Establish "difficulty" of a **problem**, e.g. how hard is 3SUM?
- Develop "optimal" algorithm.

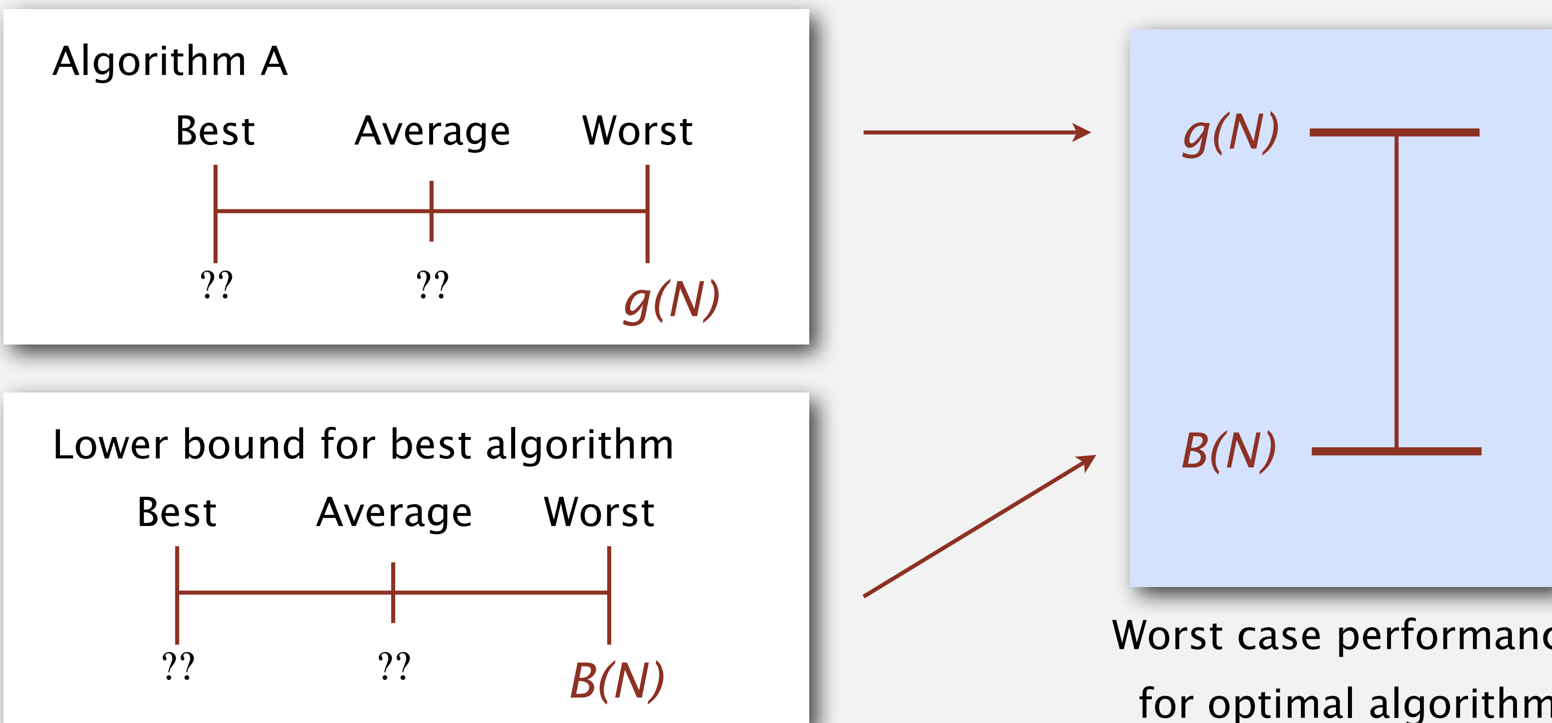
## Approach: Use order-of-growth in **worst** case

- Use order-of-growth (just like we've been doing).
  - Analysis is asymptotic, i.e. for very large N.
  - Analysis is "to within a constant factor", using OaG instead of Tilde.
- Consider only **worst case**.
  - Analysis avoids messy input models.
  - Analysis focuses on guarantees.

# Theory of algorithms

## Testing optimality of algorithm A for problem P

- Find worst case order of growth guarantee for specific algorithm A, *g(N)*
- Find lower bound on guarantee for any algorithm that solves P, *B(N)*
- If they match, i.e. *g(N) = B(N)*, then:
  - Worst case performance of A is asymptotically optimal.
  - Optimal algorithm for P has order of growth *g(N)*
- If they don't, *g(N)* at least provides an upper bound.

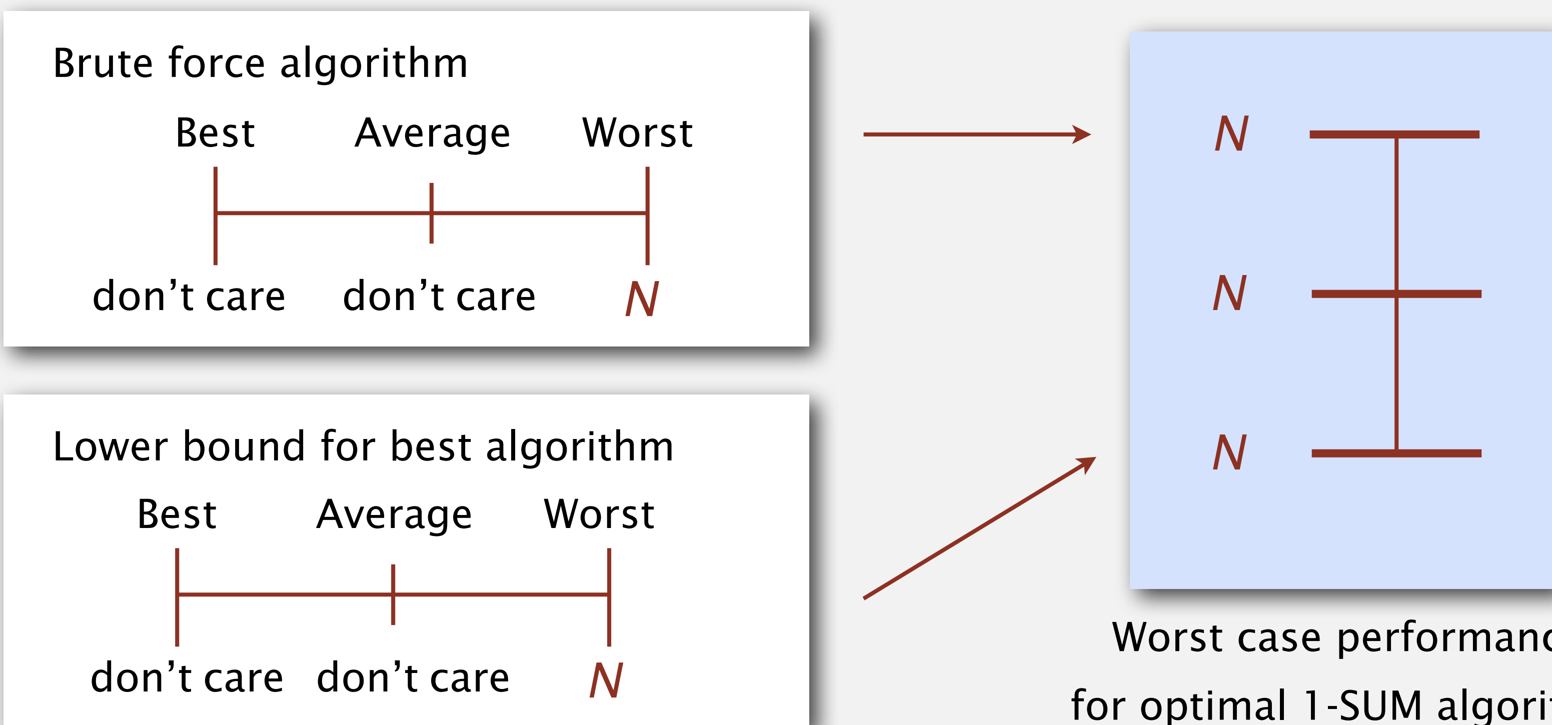



Worst case performance for optimal algorithm

# Theory of algorithms

Example: The 1-SUM problem (how many 0s?)

- Let A be the brute force algorithm where we simply look at each entry and count the zeros.
  - Worst case order of growth: $g(N) = N$
- Of any algorithm that solves 1-SUM, must at least examine every entry.
  - Lower bound on worst case order of growth: $B(N) = N$
- $g(N) = B(N)$. A is optimal!

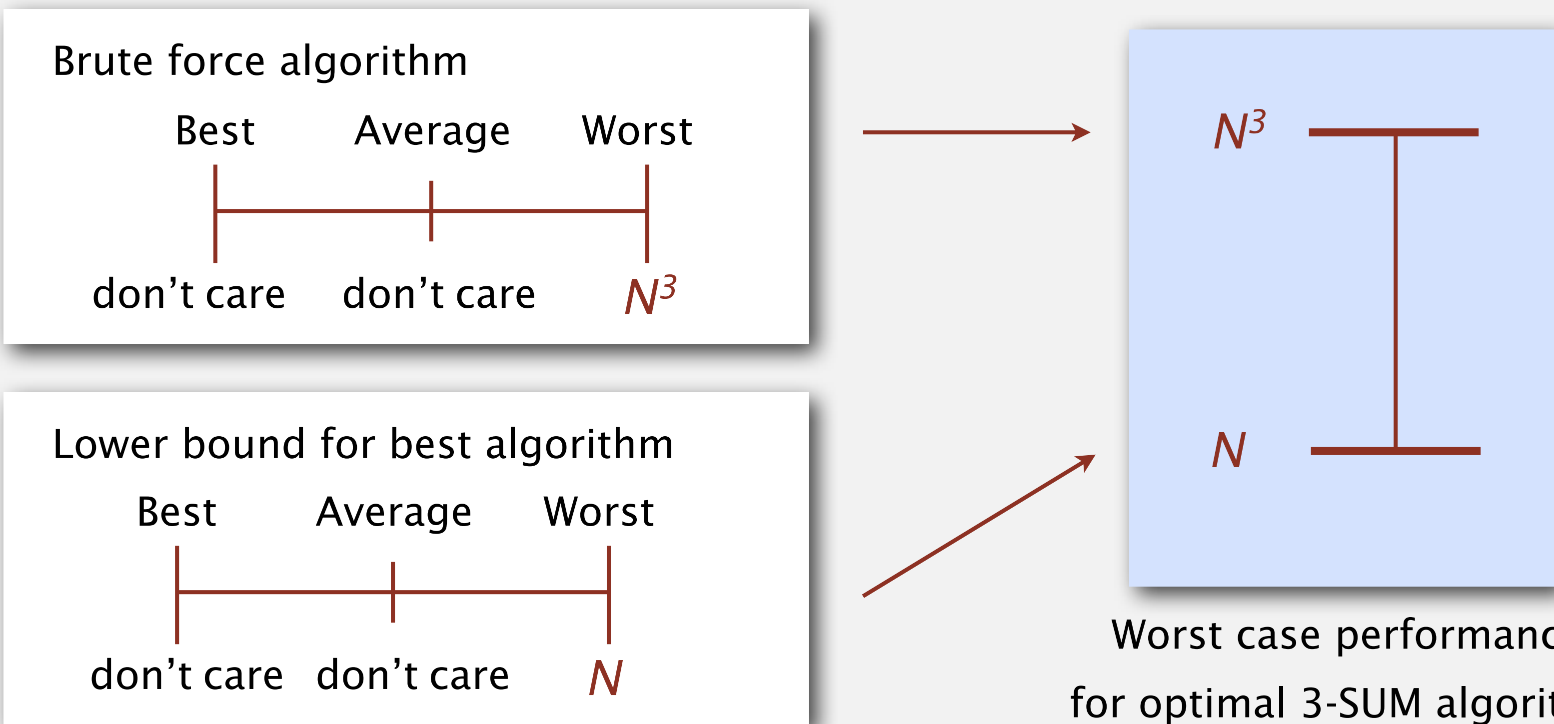



Worst case performance
for optimal 1-SUM algorithm

# Theory of algorithms: example 2

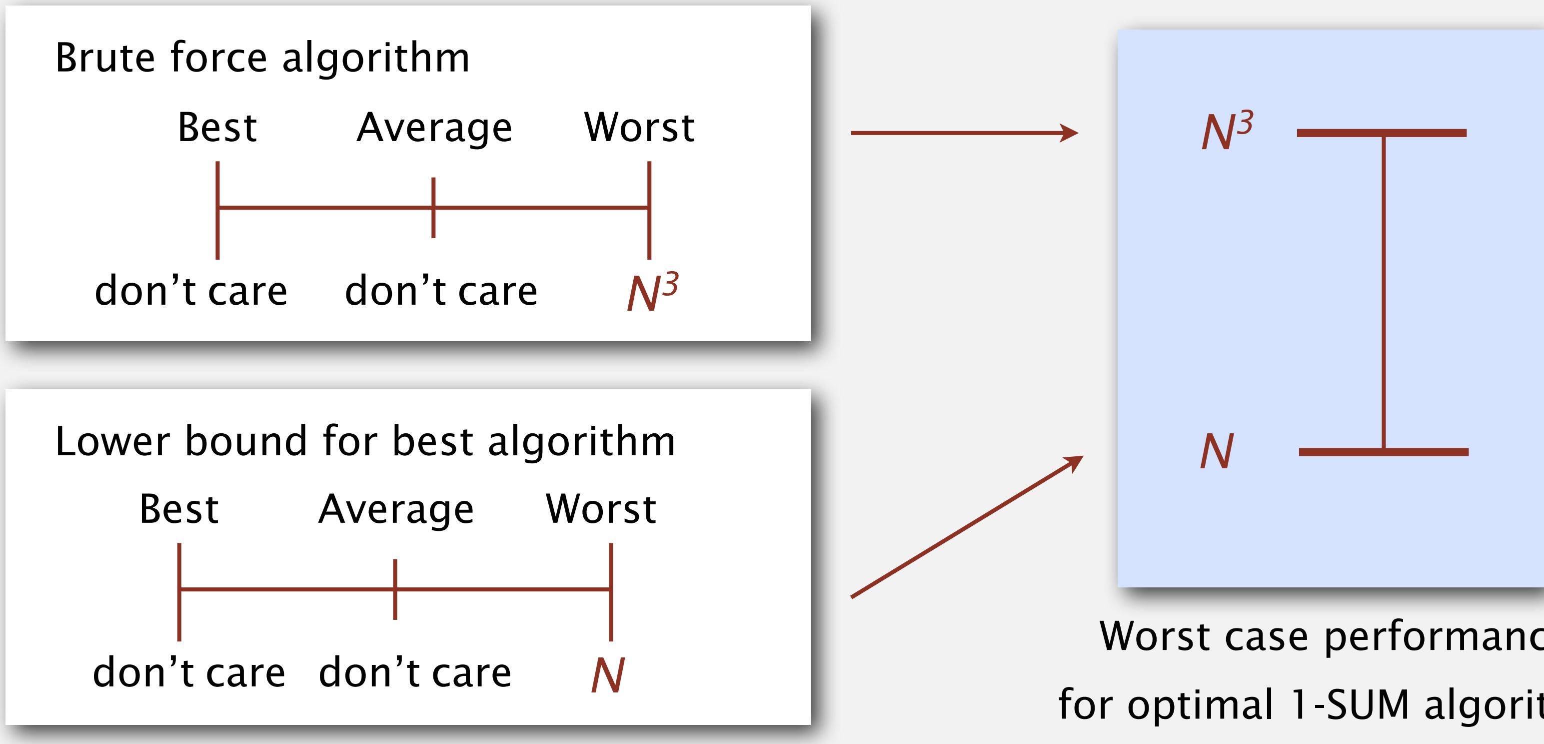
Example: The 3-SUM problem (how many 0s?)

- Let A be the brute force algorithm where we look at each triple.
  - Worst case order of growth: $g(N) = N^3$
- Of any algorithm that solves 3-SUM, must at least examine every entry.
  Lower bound on worst case order of growth: $B(N) = N$
- $g(N) \neq B(N)$




Worst case performance
for optimal 3-SUM algorithm

# Theory of algorithms: example 2




Worst case performance
for optimal 1-SUM algorithm

## What this tells us

- It is possible to solve 3SUM in $N^3$ time in the worst case.
  - The optimal algorithm has worst case running time OaG between N and $N^3$.

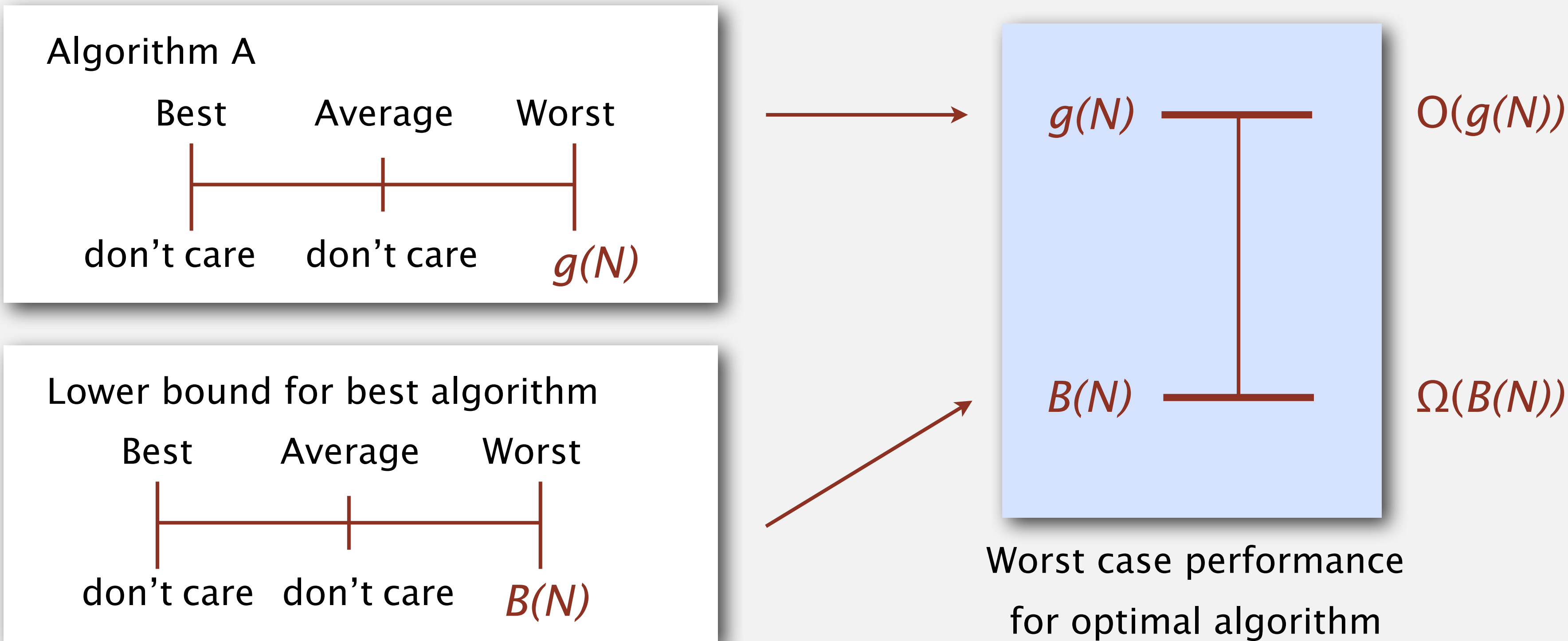
## What this doesn't tell us

- Is there an algorithm with better running time than $N^3$ in the worst case?
- Is there some clever way of finding a better lower bound than N?

# Theory of algorithms terminology

## Testing optimality of algorithm A for problem P

- Find worst case order of growth guarantee for specific algorithm A, $g(N)$
- Find lower bound on guarantee for any algorithm that solves P, $B(N)$




Worst case performance
for optimal algorithm

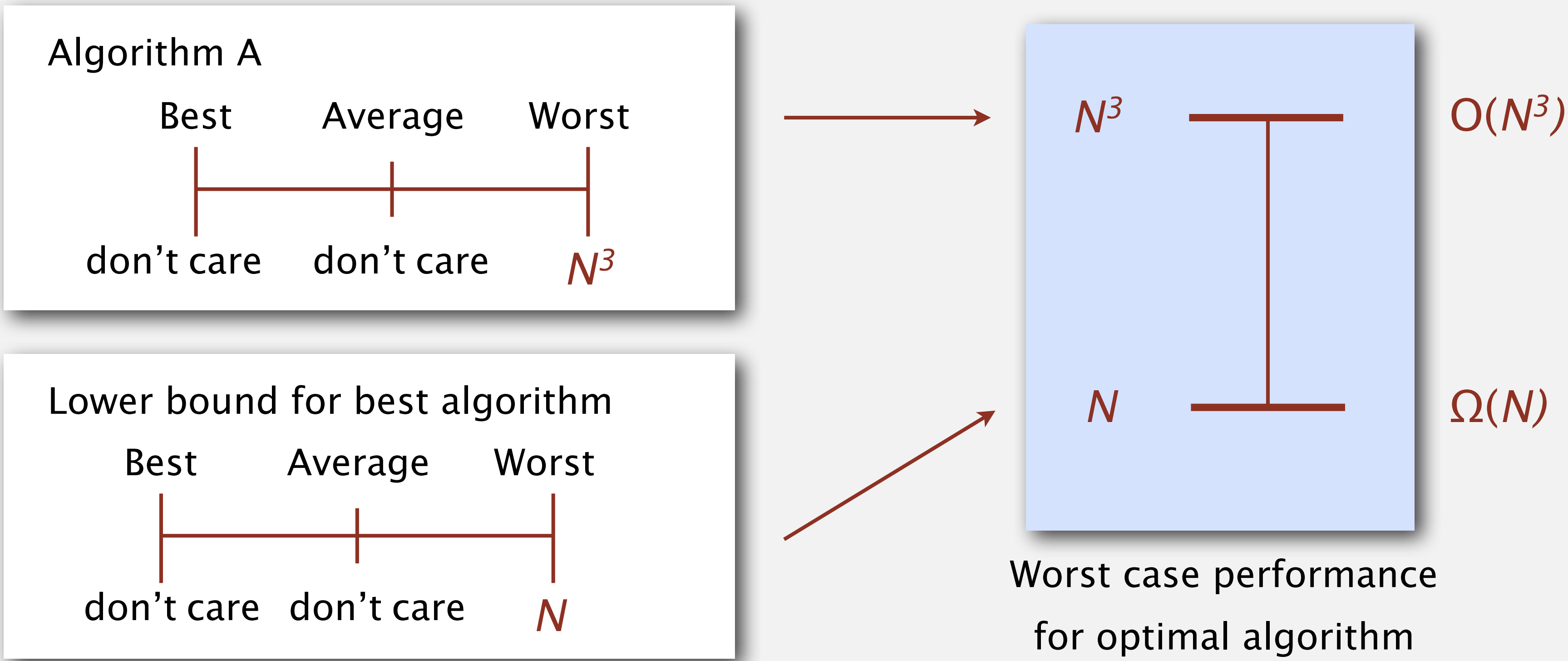
## Standard terminology

- The running time of the optimal algorithm for problem P is $O(g(N))$
- The running time of the optimal algorithm for problem P is $\Omega(B(N))$

# Theory of algorithms terminology

## Testing optimality of brute force algorithm for 3-SUM

- Find worst case order of growth guarantee for brute force, $N^3$
- Find lower bound on guarantee for any algorithm that solves P, $N$




Worst case performance
for optimal algorithm

## Standard terminology

- The running time of the optimal algorithm for problem P is $O(N^3)$
- The running time of the optimal algorithm for problem P is $\Omega(N)$

# Commonly-used notations in the theory of algorithms

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

Example: Optimal algorithm of 3-SUM is $O(N^3)$ based on brute force solution.
(i.e. its order of growth is $N^3$ or less)

# Theory of algorithm: 3SUM

## The run time of 3SUM's optimal solution...

- $=O(N^3)$ based on brute force solution.
- $=O(N^2 \log N)$ based on binary search based solution.
- $=O(N^2)$ based on solution developed in precept this week.
- $=\Omega(N)$ based on a simple argument about accessing all data.
- Grows at least as slow as $N^2$, and at least as fast as N.

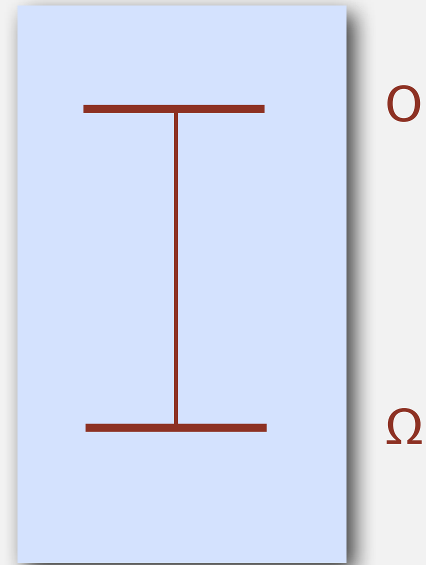
## Open questions

- What is the order of growth of the optimal solution for 3-SUM?
  - Equivalent question: If it is $\Theta(B(n))$ in the worst case, what is $B(n)$?
  - We know it is between N and $N^2$.
- Does there exist an algorithm with worst case run time better than $N^2$?
  - i.e. an algorithm that is better than $\Theta(N^2)$ in the worst case?
- Does there exist a way to provide a quadratic lower bound on 3SUM?
  - i.e. can we prove that the optimal algorithm for 3-SUM is $\Omega(N^2)$?

# Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.




Worst case performance for optimal algorithm

Gap?

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

Golden Age of Algorithm Design (1970s-Present*).

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

Caveats.

- Overly pessimistic to focus on worst case?
- Can do better than "to within a constant factor" to predict performance.
- Asymptotic performance not always useful (e.g. matrix multiplication).

# To-within a constant factor

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

Common practice. Analysis to within a constant factor.

Easy to be more precise. Use Tilde-notation instead of Big Theta (or Big Oh).

# Order of growth isn't everything - Matrix multiplication

| year | algorithm | order of growth |
|---|---|---|
| ? | brute force | $N^{3}$ |
| 1969 | Strassen | $N^{2.808}$ |
| 1978 | Pan | $N^{2.796}$ |
| 1979 | Bini | $N^{2.780}$ |
| 1981 | Schönhage | $N^{2.522}$ |
| 1982 | Romani | $N^{2.517}$ |
| 1982 | Coppersmith-Winograd | $N^{2.496}$ |
| 1986 | Strassen | $N^{2.479}$ |
| 1989 | Coppersmith-Winograd | $N^{2.376}$ |
| 2010 | Strother | $N^{2.3737}$ |
| 2011 | Williams | $N^{2.3727}$ |

Only faster for huge matrices!

**number of floating-point operations to multiply two N-by-N matrices**

- Asymptotic performance not always useful (e.g. matrix multiplication).


http://algs4.cs.princeton.edu

# 1.4 Analysis of Algorithms

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

# Basics

Bit. 0 or 1.

Byte. 8 bits.

hard drives
NIST

most computer scientists

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

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


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 |

for primitive types

| type | bytes |
| --- | --- |
| `char[]` | 2N + 24 |
| `int[]` | 4N + 24 |
| `double[]` | 8N + 24 |

for one-dimensional arrays

| type | bytes |
| --- | --- |
| `char[][]` | ~ 2 M N |
| `int[][]` | ~ 4 M N |
| `double[][]` | ~ 8 M N |

for two-dimensional arrays

# Typical memory usage for objects in Java

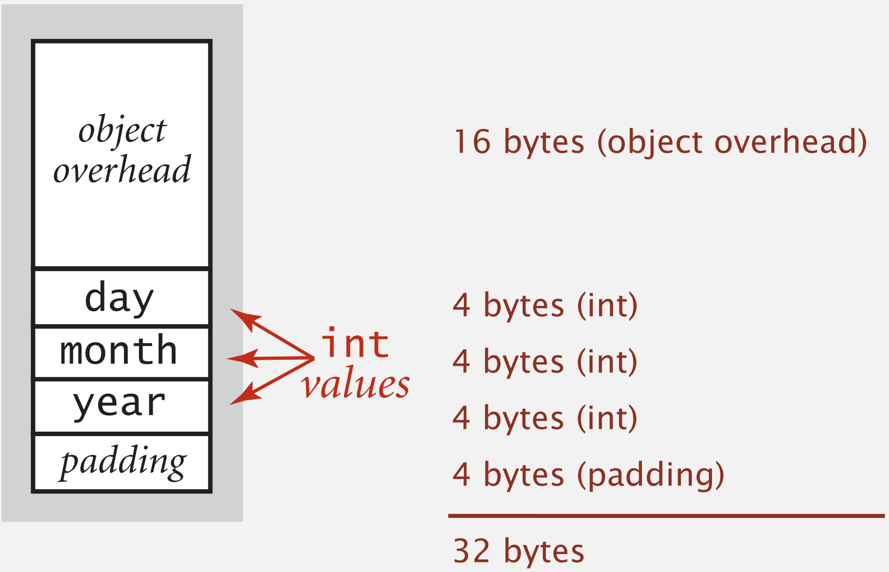
Object overhead. 16 bytes (+8 if inner class).

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
  + 8 bytes if inner class (for pointer to enclosing class).
- Padding: round up to multiple of 8 bytes.

Shallow memory usage: Don't count referenced objects.

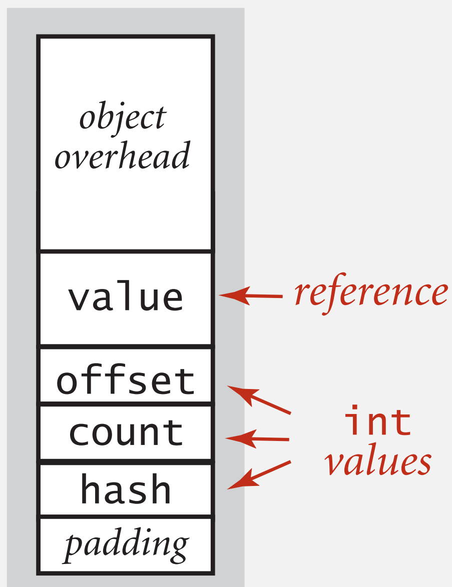
Deep memory usage: If array entry or instance variable is a reference, add memory (recursively) for referenced object.

# Typical memory usage for objects in Java

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
  + 8 bytes if inner class (for pointer to enclosing class).
- Padding: round up to multiple of 8 bytes.

```
public class String
{
   private char[] value;
   private int offset;
   private int count;
   private int hash;
...
}
```




16 bytes (object overhead)

8 bytes (reference to array)
2N + 24 bytes (char[] array)

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding)

Deep memory: 2N + 64 bytes
~2N bytes

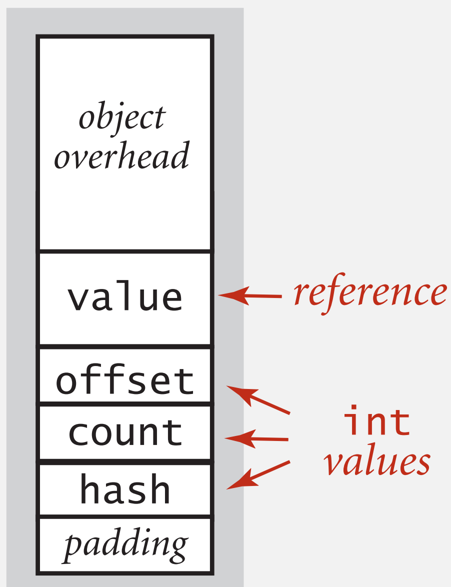
Ex 2. A `Java 6` string object uses ~2N bytes (deep).

# Typical memory usage for objects in Java

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
  + 8 bytes if inner class (for pointer to enclosing class).
- Padding: round up to multiple of 8 bytes.

```
public class String
{
   private char[] value;
   private int offset;
   private int count;
   private int hash;
...
}
```




16 bytes (object overhead)

8 bytes (reference to array)
~~2N + 24 bytes (char[] array)~~
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)

**Shallow** memory: 40 bytes

Ex 2. A `Java 6` string object uses 40 bytes (shallow memory).

# 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) each reference + int[] array

4 bytes (int)

4 bytes (padding)

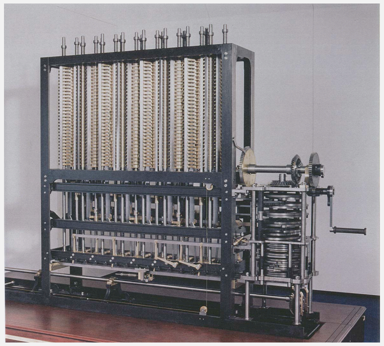
A. $8N + 88 \sim 8N$ bytes.

# Turning the crank: summary

Empirical analysis.

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

Mathematical analysis.



- Analyze algorithm to count frequency of operations.
- Use tilde notation or order of growth 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.