Section 4.1

7. Performance
7. Performance

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

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

Difference Engine #2
Designed by Charles Babbage, c. 1848
Built by London Science Museum, 1991
The challenge (modern version)

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

Review: program development in the real world

A four-step process, with feedback.

EDIT your program.

COMPILE your program to create an executable file.

RUN your program to test that it works as you imagined.

TEST your program on realistic and real input data.

USE your program to solve a 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.

```java
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.*
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 $2^x = N$.

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

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

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

**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.

**Frequently encountered values**

<table>
<thead>
<tr>
<th>$N$</th>
<th>approximate value</th>
<th>$\lg N$</th>
<th>$\log_{10} N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>1 thousand</td>
<td>10</td>
<td>3.01</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>1 million</td>
<td>20</td>
<td>6.02</td>
</tr>
<tr>
<td>$2^{30}$</td>
<td>1 billion</td>
<td>30</td>
<td>9.03</td>
</tr>
</tbody>
</table>

Prove by induction. Details in "sorting and searching" lecture.
An algorithmic challenge: 3-sum problem

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

For simplicity, just count them.

```java
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));
    }
}
```

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.

```java
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;
}
```

Q. How much time will this program take for \( N = 1 \) million?
Image sources

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:Hubble's_Wide_View_of_'Mystic_Mountain'_in_Infrared.jpg
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

```java
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

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

not much chance of a 3-sum

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

```
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);
```

Replace `println()` in `ThreeSum` with this code.

Tabulate and plot results

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>31</td>
</tr>
<tr>
<td>8000</td>
<td>248</td>
</tr>
</tbody>
</table>

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)
```
Aside: experimentation in CS

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

**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.

Do the math
- \(\lg T_N = \lg a + 3\lg N\)
- \(T_N = aN^3\)
- \(248 = a \times 8000^3\)
- \(a = 4.84 \times 10^{-10}\)
- \(T_N = 4.84 \times 10^{-10} \times N^3\)

<table>
<thead>
<tr>
<th>(N)</th>
<th>(T_N)</th>
<th>(\lg N)</th>
<th>(\lg T_N)</th>
<th>(4.84 \times 10^{-10} \times N^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.5</td>
<td>10</td>
<td>-1</td>
<td>0.5</td>
</tr>
<tr>
<td>2000</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>31</td>
<td>12</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>8000</td>
<td>248</td>
<td>13</td>
<td>8</td>
<td>248</td>
</tr>
</tbody>
</table>
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.

![Image of a crystal ball with a checkmark]

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).

![Image of a calculator converting 484 million seconds to years]

484 million seconds = 15.3374 years
Another hypothesis

1970s

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

2010s: 10,000+ times faster

VAX 11/780

Macbook Air

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

4.8 \times 10^{-10} \times N^3 \text{ seconds}

\begin{array}{|c|c|}
\hline
N & \text{time (seconds)} \\
\hline
1000 & 5319 \\
2000 & 43221 \\
4000 & 343774 \\
8000 & 2654384 \\
\hline
\end{array}

\begin{array}{|c|c|}
\hline
N & \text{time (seconds)} \\
\hline
1000 & 0 \\
2000 & 4 \\
4000 & 31 \\
8000 & 248 \\
\hline
\end{array}

(estimated)
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
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.


• 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.
**Warmup: 1-sum**

```java
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;
}
```

**Q. Formula for total running time?**

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>function call/return</td>
<td>20 ns</td>
<td>1</td>
</tr>
<tr>
<td>variable declaration</td>
<td>2 ns</td>
<td>2</td>
</tr>
<tr>
<td>assignment</td>
<td>1 ns</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2}$ ns</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2}$ ns</td>
<td>$N$</td>
</tr>
<tr>
<td>array access</td>
<td>$\frac{1}{2}$ ns</td>
<td>$N$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2}$ ns</td>
<td>between $N$ and $2N$</td>
</tr>
</tbody>
</table>

Note that frequency of increments depends on input.

Representative estimates (with some poetic license); knowing exact values may require study and experimentation.
Warmup: 2-sum

```java
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;
}
```

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>function call/return</td>
<td>20 ns</td>
<td>1</td>
</tr>
<tr>
<td>variable declaration</td>
<td>2 ns</td>
<td>N + 2</td>
</tr>
<tr>
<td>assignment</td>
<td>1 ns</td>
<td>N + 2</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/2 ns</td>
<td>(N + 1) (N + 2)/2</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/2 ns</td>
<td>N (N − 1)/2</td>
</tr>
<tr>
<td>array access</td>
<td>1/2 ns</td>
<td>N (N − 1)</td>
</tr>
<tr>
<td>increment</td>
<td>1/2 ns</td>
<td>between N (N + 1)/2 and N^2</td>
</tr>
</tbody>
</table>

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 $f(N)/g(N) \rightarrow 1$ as $N \rightarrow \infty$

**Ex.** $\frac{5}{4} N^2 + \frac{13}{4} N + \frac{53}{2} \sim \frac{5}{4} N^2$

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

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

A. $\sim \frac{5}{4} N^2$ nanoseconds.

eliminate dependence on input
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;
}

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

A. $\sim N^3/2$ nanoseconds.

### Table

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>function call/return</td>
<td>20 ns</td>
<td>1</td>
</tr>
<tr>
<td>variable declaration</td>
<td>2 ns</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>assignment</td>
<td>1 ns</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/2 ns</td>
<td>$\sim N^3/6$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/2 ns</td>
<td>$\sim N^3/6$</td>
</tr>
<tr>
<td>array access</td>
<td>1/2 ns</td>
<td>$\sim N^3/2$</td>
</tr>
<tr>
<td>increment</td>
<td>1/2 ns</td>
<td>$\sim N^3/6$</td>
</tr>
</tbody>
</table>

# $i < j < k = \binom{N}{3} = \frac{N(N-1)(N-2)}{6} \sim \frac{N^3}{6}$ assumes count is not large

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

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.
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
7. Performance

- The challenge
- Empirical analysis
- Mathematical models
- Doubling method
- Familiar examples
Key questions and answers

Q. Is the running time of my program \( \sim a N^b \) 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 $2^b$.

**Proof:**

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

**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**

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$</th>
<th>$T_N/T_{N/2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4000</td>
<td>31</td>
<td>7.75</td>
</tr>
<tr>
<td>8000</td>
<td>248</td>
<td>8</td>
</tr>
<tr>
<td>16000</td>
<td>$248 \times 8 = 1984$</td>
<td>8</td>
</tr>
<tr>
<td>32000</td>
<td>$248 \times 8^2 = 15872$</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024000</td>
<td>$248 \times 8^7 = 520093696$</td>
<td>8</td>
</tr>
</tbody>
</table>

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

**Def.** If a function \( f(N) \sim ag(N) \) 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**

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**

Machine- and system-dependent features of the model are all constants.
Hypothesis. The order of growth of the running time of my program is $N^b (\log N)^c$.

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

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

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

**Quadratic (N²)**
```
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        ...
```

**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++)
        ...
}
```

**Cubic (N³)**
```
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
    for (int k = j+1; k < N; k++)
        ...
```

**Exponential (2^N)**
```
public static void f(int N)
{
    if (N == 0) return;
    ... f(N-1)...
    ... f(N-1)...
}
```

Stay tuned for examples.
Order of growth classifications

<table>
<thead>
<tr>
<th>order of growth</th>
<th>description</th>
<th>function</th>
<th>slope of line in log-log plot (b)</th>
<th>factor for doubling method ($2^b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>logarithmic</td>
<td>logN</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>linear</td>
<td>N</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>linearithmic</td>
<td>N logN</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>quadratic</td>
<td>$N^2$</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>cubic</td>
<td>$N^3$</td>
<td>3</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

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

If input size doubles running time increases by this factor.

Math model may have log factor.
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**

\[
T_N = aN^3 \quad \text{running time today}
\]

\[
T_{2N} = \left(\frac{a}{2}\right)(2N)^3 \quad \text{running time in 2 years}
\]

\[
= 4aN^3
\]

\[
= 4T_N
\]

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

There are many other apps running on my computer!

Your *input* model is too simple: My real input data is completely different.

What happens when the leading term oscillates?

Your *machine* model is too simple: My computer has parallel processors and a cache.

Where’s the log factor?

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

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

```java
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)
", now - start);
    }
}
```

A. 4.8 million seconds (about 2 months).

<table>
<thead>
<tr>
<th>N</th>
<th>T_N</th>
<th>T_N / T_N/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>17</td>
<td>4.25</td>
</tr>
<tr>
<td>4000</td>
<td>56</td>
<td>3.29</td>
</tr>
<tr>
<td>8000</td>
<td>286</td>
<td>5.10</td>
</tr>
<tr>
<td>16000</td>
<td>1172</td>
<td>4.09</td>
</tr>
<tr>
<td>32000</td>
<td>1172 * 4 = 4688</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1024000</td>
<td>1172 * 4^6 = 4800512</td>
<td>4</td>
</tr>
</tbody>
</table>

% 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)

math model says order of growth should be N^2

✓
Pop quiz on performance

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

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$ (in seconds)</th>
<th>$T_N/T_{N/2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>8000</td>
<td>320</td>
<td>4</td>
</tr>
</tbody>
</table>

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]);
        ...
    }
}
```

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$ (in seconds)</th>
<th>$T_N/T_{N/2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>8000</td>
<td>320</td>
<td>4</td>
</tr>
<tr>
<td>16000</td>
<td>$320 \times 4 = 1280$</td>
<td>4</td>
</tr>
<tr>
<td>32000</td>
<td>$1280 \times 4 = 5120$</td>
<td>4</td>
</tr>
<tr>
<td>64000</td>
<td>$5120 \times 4 = 20480$</td>
<td>4</td>
</tr>
</tbody>
</table>

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

**A.** 20480 seconds.

**Q.** Estimate the order of growth.

**A.** $N^2$, since $\log 4 = 2$. 
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 %d %.0f ", N, N*Math.log(N) + .57721*N);
        StdOut.print(cardcnt/trials);
        double now = System.currentTimeMillis()/1000.0;
        StdOut.printf("%d %.0f seconds\n", now - start);
    }
}
```

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

Q. How long to simulate collecting 1 million coupons?

<table>
<thead>
<tr>
<th>N</th>
<th>$T_N$</th>
<th>$T_N/T_N/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>125000</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>250000</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>500000</td>
<td>31</td>
<td>2.21</td>
</tr>
<tr>
<td>1000000</td>
<td>$31 \times 2 = 63$</td>
<td>2</td>
</tr>
</tbody>
</table>

% 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)

math model says order of growth should be $N\log N$
Analyzing typical memory requirements

A **bit** is 0 or 1 and the basic unit of memory. 1 **megabyte** (MB) is about 1 million bytes. 1 **gigabyte** (GB) is about 1 billion bytes.

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

### Primitive-type values

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: **not** 1 bit

### System-supported data structures (typical)

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[N]</td>
<td>4N + 16</td>
</tr>
<tr>
<td>double[N]</td>
<td>8N + 16</td>
</tr>
<tr>
<td>int[N][N]</td>
<td>4N^2 + 20N + 16 ~ 4N^2</td>
</tr>
<tr>
<td>double[N][N]</td>
<td>8N^2 + 20N + 16 ~ 8N^2</td>
</tr>
<tr>
<td>String</td>
<td>2N + 40</td>
</tr>
</tbody>
</table>

Example. 2000-by-2000 double array uses ~32MB.
Use computational experiments, mathematical analysis, and the \textit{scientific method} to learn whether your program might be useful to solve a large problem.

\textbf{Q. What if it's not fast enough?}

\textbf{A.}

- **Yes**: \textit{Does it scale?}
  - **Yes**: \textit{Buy a new computer and solve bigger problems}
  - **No**: \textit{Learn a better algorithm}

- **No**: \textit{Found one?}
  - **No**: \textit{Invent a better algorithm}
  - **Yes**: \textit{ Plenty of new algorithms awaiting discovery. Example: Solve 3-sum efficiently}
Case in point

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

Lesson. Performance matters!

Invent a better algorithm

PageRank (see Section 1.6)
Image source

7. Performance