1.4 Analysis of Algorithms

- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory
Cast of characters

**Programmer** needs to develop a working solution.

**Client** wants to solve problem efficiently.

**Theoretician** wants to understand.

**Student** might play any or all of these roles someday.

**Basic blocking and tackling is sometimes necessary.**
[This lecture]
“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)
Reasons to analyze algorithms

- Predict performance.
- Compare algorithms.  
  this course (COS 226)
- Provide guarantees.
- Understand theoretical basis.  
  theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs.

Client gets poor performance because programmer did not understand performance characteristics
Some algorithmic successes

Discrete Fourier transform.

- Break down waveform of $N$ samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics, ....
- Brute force: $N^2$ steps.
- FFT algorithm: $N \log N$ steps, enables new technology.
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.
The challenge

Q. Will my program be able to solve a large practical input?

- Why is my program so slow?
- Why does it run out of memory?

Key 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.
- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>-40</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>-20</td>
<td>-10</td>
</tr>
<tr>
<td>3</td>
<td>-40</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Context. Deeply related to problems in computational geometry.
public class ThreeSum
{
   public static int count(int[] a)
   {
      int N = a.length;
      int count = 0;
    ...
   }
}

public static void main(String[] args)
{
   int[] a = In.readInts(args[0]);
   StdOut.println(count(a));
}
Measuring the running time

Q. How to time a program?
A. Manual.
Measuring the running time

Q. How to time a program?
A. Automatic.

```java
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();
}
```

**public class Stopwatch** *(part of stdlib.jar)*

- `Stopwatch()` *create a new stopwatch*
- `double elapsedTime()` *time since creation (in seconds)*
Empirical analysis

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

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>?</td>
</tr>
</tbody>
</table>
Data analysis

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

![Standard plot](image)

- **Data analysis**

- **1K**
- **2K**
- **4K**
- **8K**

- **Log-log plot**

- **Running time** $T(N)$

- **Problem size** $N$

- **Straight line of slope 3**
Data analysis

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

Regression. Fit straight line through data points: $a N^b$.

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

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

**Predictions.**
- 51.0 seconds for $N = 8,000$.
- 408.1 seconds for $N = 16,000$.

**Observations.**

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,000</td>
<td>51.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>410.8</td>
</tr>
</tbody>
</table>

"order of growth" of running time is about $N^3$ [stay tuned] validates hypothesis!
Doubling hypothesis

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

Run program, doubling the size of the input.

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
<th>ratio</th>
<th>lg ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
<td>4.8</td>
<td>2.3</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
<td>6.9</td>
<td>2.8</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>7.7</td>
<td>2.9</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>8.0</td>
<td>3.0</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>8.0</td>
<td>3.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

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 hypothesis.

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

**A.** Run the program (for a sufficient large value of $N$) and solve for $a$.

\[
\begin{array}{|c|c|}
\hline
N & \text{time (seconds)} \dagger \\
\hline
8,000 & 51.1 \\
\hline
8,000 & 51.0 \\
\hline
8,000 & 51.1 \\
\hline
\end{array}
\]

\[
51.1 = a \times 8000^3
\]

\[
\Rightarrow a = 0.998 \times 10^{-10}
\]

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

almost identical hypothesis to one obtained via linear regression
Experimental algorithmics

System independent effects.
• Algorithm.
• Input data.

System dependent effects.
• Hardware: CPU, memory, cache, …
• Software: compiler, interpreter, garbage collector, …
• System: operating system, network, other applications, …

Bad news. Difficult to get precise measurements.
Good news. Much easier and cheaper than other sciences.

\[ a = \text{determines constant in power law} \]
\[ b = \text{determines exponent in power law} \]

\text{e.g., can run huge number of experiments}
War story (from COS 126)

Q. How long does this program take as a function of $N$?

```java
String s = StdIn.readString();
int N = s.length();
...
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
       distance[i][j] = ...
...
```

<table>
<thead>
<tr>
<th>$N$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.11</td>
</tr>
<tr>
<td>2,000</td>
<td>0.35</td>
</tr>
<tr>
<td>4,000</td>
<td>1.6</td>
</tr>
<tr>
<td>8,000</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Jenny $\sim c_1 N^2$ seconds

<table>
<thead>
<tr>
<th>$N$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.5</td>
</tr>
<tr>
<td>500</td>
<td>1.1</td>
</tr>
<tr>
<td>1,000</td>
<td>1.9</td>
</tr>
<tr>
<td>2,000</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Kenny $\sim c_2 N$ seconds
› observations
› mathematical models
› order-of-growth classifications
› dependencies on inputs
› memory
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.

In principle, accurate mathematical models are available.
# Cost of basic operations

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds †</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer add</td>
<td>a + b</td>
<td>2.1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>a * b</td>
<td>2.4</td>
</tr>
<tr>
<td>integer divide</td>
<td>a / b</td>
<td>5.4</td>
</tr>
<tr>
<td>floating-point add</td>
<td>a + b</td>
<td>4.6</td>
</tr>
<tr>
<td>floating-point multiply</td>
<td>a * b</td>
<td>4.2</td>
</tr>
<tr>
<td>floating-point divide</td>
<td>a / b</td>
<td>13.5</td>
</tr>
<tr>
<td>sine</td>
<td>Math.sin(theta)</td>
<td>91.3</td>
</tr>
<tr>
<td>arctangent</td>
<td>Math.atan2(y, x)</td>
<td>129.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM
Cost of basic operations

| operation                        | example                  | nanoseconds  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>int a</td>
<td>$c_1$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>a = b</td>
<td>$c_2$</td>
</tr>
<tr>
<td>integer compare</td>
<td>a &lt; b</td>
<td>$c_3$</td>
</tr>
<tr>
<td>array element access</td>
<td>a[i]</td>
<td>$c_4$</td>
</tr>
<tr>
<td>array length</td>
<td>a.length</td>
<td>$c_5$</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[N]</td>
<td>$c_6 N$</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[N][N]</td>
<td>$c_7 N^2$</td>
</tr>
<tr>
<td>string length</td>
<td>s.length()</td>
<td>$c_8$</td>
</tr>
<tr>
<td>substring extraction</td>
<td>s.substring(N/2, N)</td>
<td>$c_9$</td>
</tr>
<tr>
<td>string concatenation</td>
<td>s + t</td>
<td>$c_{10} N$</td>
</tr>
</tbody>
</table>

Novice mistake. Abusive string concatenation.
Q. How many instructions as a function of input size $N$?

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

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$N$</td>
</tr>
<tr>
<td>array access</td>
<td>$N$</td>
</tr>
<tr>
<td>increment</td>
<td>$N$ to $2N$</td>
</tr>
</tbody>
</table>
Example: 2-sum

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

```java
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++;
```

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
</tr>
</tbody>
</table>

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

tedious to count exactly
“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
Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```c
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++;
```

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

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>N + 2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>N + 2</td>
</tr>
<tr>
<td>less than compare</td>
<td>(\frac{1}{2} (N + 1) (N + 2))</td>
</tr>
<tr>
<td>equal to compare</td>
<td>(\frac{1}{2} N (N - 1))</td>
</tr>
<tr>
<td>array access</td>
<td>N (N - 1)</td>
</tr>
<tr>
<td>increment</td>
<td>(\frac{1}{2} N (N - 1)) to N (N - 1)</td>
</tr>
</tbody>
</table>

cost model = array accesses
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. $\frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N \sim \frac{1}{6} N^3$

Simplification 2: tilde notation

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

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>tilde notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1)(N + 2)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
<td>$\sim N^2$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$ to $\sim N^2$</td>
</tr>
</tbody>
</table>
Example: 2-sum

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

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

Bottom line. Use cost model and tilde notation to simplify frequency counts.
Example: 3-sum

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

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

```
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++;
```

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

$\sim \frac{1}{6} N^3$

Bottom line. Use cost model and tilde notation to simplify frequency counts.
Estimating a discrete sum

Q. How to estimate a discrete sum?
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. $1 + 1/2 + 1/3 + \ldots + 1/N$.

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

Ex 3. 3-sum triple loop.

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

In principle, accurate mathematical models are available.

In practice,
• Formulas can be complicated.
• Advanced mathematics might be required.
• Exact models best left for experts.

Bottom line. We use approximate models in this course: \( T(N) \sim c N^3 \).
• observations
• mathematical models
• order-of-growth classifications
• dependencies on inputs
• memory
Good news. the small set of functions

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

suffices to describe order-of-growth of typical algorithms.
## Common order-of-growth classifications

<table>
<thead>
<tr>
<th>order of growth</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
<th>T(2N) / T(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td><code>a = b + c;</code></td>
<td>statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td><code>while (N &gt; 1) { N = N / 2; ... }</code></td>
<td>divide in half</td>
<td>binary search</td>
<td>~ 1</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td><code>for (int i = 0; i &lt; N; i++) { ... }</code></td>
<td>loop</td>
<td>find the maximum</td>
<td>2</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>[see mergesort lecture]</td>
<td>divide and conquer</td>
<td>mergesort</td>
<td>~ 2</td>
</tr>
</tbody>
</table>
| N^2             | quadratic  | ```for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++)
{ ... }```  | double loop          | check all pairs         | 4            |
| N^3             | cubic      | ```for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++)
for (int k = 0; k < N; k++)
{ ... }```  | triple loop          | check all triples       | 8            |
| 2^N             | exponential| [see combinatorial search lecture]       | exhaustive search    | check all subsets        | T(N)         |
**Practical implications of order-of-growth**

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1970s</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
</tr>
<tr>
<td>N²</td>
<td>hundreds</td>
</tr>
<tr>
<td>N³</td>
<td>hundred</td>
</tr>
<tr>
<td>2ᴺ</td>
<td>20</td>
</tr>
</tbody>
</table>

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

**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.
Binary search demo
Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946; first bug-free one published in 1962.
- Java bug in `Arrays.binarySearch()` discovered in 2006.

```java
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;
}
```

Invariant. If `key` appears in the array `a[]`, then `a[lo] ≤ key ≤ a[hi].`
Binary search: mathematical analysis

**Proposition.** Binary search uses at most $1 + \lg N$ compares to search in a sorted array of size $N$.

**Def.** $T(N) \equiv \#$ compares to binary search in a sorted subarray of size at most $N$.

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

**Pf sketch.**

\[
T(N) \leq T(N/2) + 1 \\
\leq T(N/4) + 1 + 1 \\
\leq T(N/8) + 1 + 1 + 1 \\
\ldots \\
\leq T(N/N) + 1 + 1 + \ldots + 1 \\
= 1 + \lg N
\]

given

apply recurrence to first term

apply recurrence to first term

stop applying, $T(1) = 1$
An $N^2 \log N$ algorithm for 3-sum

Algorithm.
- Sort the $N$ (distinct) numbers.
- 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.
Comparing programs

**Hypothesis.** The $N^2 \log N$ three-sum algorithm is significantly faster in practice than the brute-force $N^3$ algorithm.

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

ThreeSum.java

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.14</td>
</tr>
<tr>
<td>2,000</td>
<td>0.18</td>
</tr>
<tr>
<td>4,000</td>
<td>0.34</td>
</tr>
<tr>
<td>8,000</td>
<td>0.96</td>
</tr>
<tr>
<td>16,000</td>
<td>3.67</td>
</tr>
<tr>
<td>32,000</td>
<td>14.88</td>
</tr>
<tr>
<td>64,000</td>
<td>59.16</td>
</tr>
</tbody>
</table>

ThreeSumDeluxe.java

**Guiding principle.** Typically, better order of growth $\Rightarrow$ faster in practice.
• observations
• mathematical models
• order-of-growth classifications
• dependencies on inputs
• memory
Types of analyses

Best case. Lower bound on cost.
• Determined by “easiest” input.
• Provides a goal for all inputs.

Worst case. Upper bound on cost.
• Determined by “most difficult” input.
• Provides a guarantee for all inputs.

Average case. Expected cost for random input.
• Need a model for “random” input.
• Provides a way to predict performance.

Ex 1. Array accesses for brute-force 3 sum.
Best: \( \sim \frac{1}{2} N^3 \)
Average: \( \sim \frac{1}{2} N^3 \)
Worst: \( \sim \frac{1}{2} N^3 \)

Ex 2. Compares for binary search.
Best: \( \sim 1 \)
Average: \( \sim \lg N \)
Worst: \( \sim \lg N \)
### Commonly-used notations

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>~ 10 (N^2)</td>
<td>10 (N^2) 10 (N^2) + 22 (N \log N) 10 (N^2) + 2 (N + 37)</td>
<td>provide approximate model</td>
</tr>
<tr>
<td>Big Theta</td>
<td>asymptotic growth rate</td>
<td>(\Theta(N^2))</td>
<td>(\frac{1}{2} (N^2) 10 (N^2) 5 (N^2) + 22 (N \log N) + 3(N)</td>
<td>classify algorithms</td>
</tr>
<tr>
<td>Big Oh</td>
<td>(\Theta(N^2)) and smaller</td>
<td>(O(N^2))</td>
<td>10 (N^2) 100 (N) 22 (N \log N) + 3 (N)</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td>Big Omega</td>
<td>(\Theta(N^2)) and larger</td>
<td>(\Omega(N^2))</td>
<td>(\frac{1}{2} (N^2) (N^5) (N^3) + 22 (N \log N) + 3 (N)</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>

**Common mistake.** Interpreting big-Oh as an approximate model.
observations
mathematical models
order-of-growth classifications
dependencies on inputs
memory
Basics

Bit. 0 or 1.
Byte. 8 bits.
Megabyte (MB). 1 million or $2^{20}$ bytes.
Gigabyte (GB). 1 billion or $2^{30}$ bytes.

Old machine. We used to assume a 32-bit machine with 4 byte pointers.

Modern machine. We now assume a 64-bit machine with 8 byte pointers.
• Can address more memory.
• Pointers use more space.

NIST most computer scientists

some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost
## Typical memory usage for primitive types and arrays

### Primitive types.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>byte</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>

For primitive types

### Array overhead. 24 bytes.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>2N + 24</td>
</tr>
<tr>
<td>int[]</td>
<td>4N + 24</td>
</tr>
<tr>
<td>double[]</td>
<td>8N + 24</td>
</tr>
</tbody>
</table>

For one-dimensional arrays

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[][]</td>
<td>~ 2 MN</td>
</tr>
<tr>
<td>int[][]</td>
<td>~ 4 MN</td>
</tr>
<tr>
<td>double[][]</td>
<td>~ 8 MN</td>
</tr>
</tbody>
</table>

For two-dimensional arrays
Typical memory usage for objects in Java

**Object overhead.** 16 bytes.

**Reference.** 8 bytes.

**Padding.** Each object uses a multiple of 8 bytes.

**Ex 1.** A `Date` object uses 32 bytes of memory.

```java
public class Date {
    private int day;
    private int month;
    private int year;
    ...
}
```

```
<table>
<thead>
<tr>
<th>16 bytes (object overhead)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes (int)</td>
</tr>
<tr>
<td>4 bytes (int)</td>
</tr>
<tr>
<td>4 bytes (int)</td>
</tr>
<tr>
<td>4 bytes (padding)</td>
</tr>
<tr>
<td>32 bytes</td>
</tr>
</tbody>
</table>
```
**Typical memory usage for objects in Java**

**Object overhead.** 16 bytes.

**Reference.** 8 bytes.

**Padding.** Each object uses a multiple of 8 bytes.

**Ex 2.** A virgin string of length $N$ uses $\sim 2N$ bytes of memory.

```java
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)**

$2N + 64$ bytes
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 if inner class.

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.
Q. How much memory does `WeightedQuickUnionUF` use as a function of $N$? Use tilde notation to simplify your answer.
Turning the crank: summary

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

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

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