1.4 Analysis of Algorithms

- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space

Reference: Intro to Programming in Java, Section 4.1
Cast of characters

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

**Client** wants problem solved 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

Charles Babbage (1864)  
Analytic Engine
Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

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.

Friedrich Gauss
1805

![Graph showing time vs. size with linear, linearithmic, and quadratic scales.](image)
Some algorithmic successes

N-body Simulation.
- Simulate gravitational interactions among N bodies.
- Brute force: $N^2$ steps.
- Barnes-Hut: $N \log N$ steps, enables new research.

Andrew Appel
PU ’81
- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space
Scientific analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.
- **Observe** some feature of the universe.
- **Hypothesize** a model that is consistent with observation.
- **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**.

Universe = computer itself.
Every time you run a program you are doing an experiment!

First step. Debug your program!
Second step. Choose input model for experiments.
Third step. Run and time the program for problems of increasing size.
Example: 3-sum

3-sum. Given \( N \) integers, find all triples that sum to exactly zero.

```plaintext
% more input8.txt
8
 30 -30 -20 -10 40 0 10 5

% java ThreeSum < input8.txt
4
 30 -30  0
 30 -20 -10
-30 -10  40
-10   0  10
```

Context. Deeply related to problems in computational geometry.
public class ThreeSum
{
    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;
    }

    public static void main(String[] args)
    {
        long[] a = StdArrayIO.readInt1D();
        StdOut.println(count(a));
    }
}
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>1000</td>
<td>0.26</td>
</tr>
<tr>
<td>2000</td>
<td>2.16</td>
</tr>
<tr>
<td>4000</td>
<td>17.18</td>
</tr>
<tr>
<td>8000</td>
<td>137.76</td>
</tr>
</tbody>
</table>

† Running Linux on Sun-Fire-X4100
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
Stopwatch stopwatch = new Stopwatch();

ThreeSum.count(a);

double time = stopwatch.elapsedTime();
StdOut.println("Running time: " + time + " seconds");
```

client code

```java
public class Stopwatch {
   private final long start = System.currentTimeMillis();
   public double elapsedTime() {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }
}
```

implementation (part of stdlib.jar, see http://www.cs.princeton.edu/introcs/stdlib)
Data analysis

Plot running time as a function of input size $N$. 

![Plot of running time vs. input size](image)

- Time axis: $time$
- Input size axis: $size$
- Key values:
  - $1K$
  - $2K$
  - $4K$
  - $8K$

Values:
- $64T$
- $128T$
- $256T$
- $512T$
Data analysis

**Log-log plot.** Plot running time vs. input size $N$ on log-log scale.

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

Hypothesis. Running time grows with the cube of the input size: $a N^3$. 
Doubling hypothesis

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

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>500</td>
<td>0.03</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>0.26</td>
<td>7.88</td>
<td>2.98</td>
</tr>
<tr>
<td>2,000</td>
<td>2.16</td>
<td>8.43</td>
<td>3.08</td>
</tr>
<tr>
<td>4,000</td>
<td>17.18</td>
<td>7.96</td>
<td>2.99</td>
</tr>
<tr>
<td>8,000</td>
<td>137.76</td>
<td>7.96</td>
<td>2.99</td>
</tr>
</tbody>
</table>

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

Caveat. Can't identify logarithmic factors with doubling hypothesis.
Prediction and verification

**Hypothesis.** Running time is about $a N^3$ for input of size $N$.

**Q.** How to estimate $a$?

**A.** Run the program!

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,000</td>
<td>17.18</td>
</tr>
<tr>
<td>4,000</td>
<td>17.15</td>
</tr>
<tr>
<td>4,000</td>
<td>17.17</td>
</tr>
</tbody>
</table>

$$17.17 = a \times 4000^3$$

$\Rightarrow a = 2.7 \times 10^{-10}$

**Refined hypothesis.** Running time is about $2.7 \times 10^{-10} \times N^3$ seconds.

**Prediction.** 1,100 seconds for $N = 16,000$.

**Observation.**

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td>1118.86</td>
</tr>
</tbody>
</table>

validates hypothesis!
Experimental algorithmics

Many obvious factors affect running time:
• Machine.
• Compiler.
• Algorithm.
• Input data.

More factors (not so obvious):
• Caching.
• Garbage collection.
• Just-in-time compilation.
• CPU use by other applications.

Bad news. It is often difficult to get precise measurements.
Good news. Easier than other sciences.

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
public class EditDistance {
    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.

Kenny. $\sim c_2 N$ seconds.
• estimating running time
• mathematical analysis
• order-of-growth hypotheses
• input models
• measuring space
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 \times 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 \times 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

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds $^\dagger$</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.
Example: 1-sum

Q. How many instructions as a function of 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>≤ 2N</td>
</tr>
</tbody>
</table>

between N (no zeros) and 2N (all zeros)
Example: 2-sum

Q. How many instructions as a function of N?

```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++;
```

<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>(\leq N^2)</td>
</tr>
</tbody>
</table>

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

tedious to count exactly
Tilde notation

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

**Ex 1.** $6N^3 + 20N + 16 \sim 6N^3$

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

**Ex 3.** $6N^3 + 17N^2 \log N + 7N \sim 6N^3$

discard lower-order terms
(e.g., $N = 1000$: 6 billion vs. 169 million)

**Technical definition.** $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$
**Example: 2-sum**

Q. How long will it take as a function of $N$?

```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>
<th>time per op</th>
<th>total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$\sim N$</td>
<td>$c_1$</td>
<td>$\sim c_1 N$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$\sim N$</td>
<td>$c_2$</td>
<td>$\sim c_2 N$</td>
</tr>
<tr>
<td>less than comparison</td>
<td>$\sim 1/2 N^2$</td>
<td>$c_3$</td>
<td>$\sim c_3 N^2$</td>
</tr>
<tr>
<td>equal to comparison</td>
<td>$\sim 1/2 N^2$</td>
<td>$c_3$</td>
<td>$\sim c_3 N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>$\sim N^2$</td>
<td>$c_4$</td>
<td>$\sim c_4 N^2$</td>
</tr>
<tr>
<td>increment</td>
<td>$\leq N^2$</td>
<td>$c_5$</td>
<td>$\leq c_5 N^2$</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td>$\sim c N^2$</td>
</tr>
</tbody>
</table>

depends on input data
Example: 3-sum

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

```java
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++;
```

Remark. Focus on instructions in **inner loop**; ignore everything else!

\[
\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \\
\sim \frac{1}{6} N^3
\]
Bounding the sum by an integral trick

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: $T_N \sim c \ N^3$. 

\[ T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E \]
- $A =$ variable declarations
- $B =$ assignment statements
- $C =$ compare
- $D =$ array access
- $E =$ increment

Costs (depend on machine, compiler) vs. frequencies (depend on algorithm, input)
- estimating running time
- mathematical analysis
- **order-of-growth hypotheses**
- input models
- measuring space
Common order-of-growth hypotheses

To determine order-of-growth:
• Assume a power law \( T_N \sim a N^b \).
• Estimate exponent \( b \) with doubling hypothesis.
• Validate with mathematical analysis.

Ex. ThreeSumDeluxe.java

Food for precept. How is it implemented?

<table>
<thead>
<tr>
<th>( N )</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.26</td>
</tr>
<tr>
<td>2,000</td>
<td>2.16</td>
</tr>
<tr>
<td>4,000</td>
<td>17.18</td>
</tr>
<tr>
<td>8,000</td>
<td>137.76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( N )</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.43</td>
</tr>
<tr>
<td>2,000</td>
<td>0.53</td>
</tr>
<tr>
<td>4,000</td>
<td>1.01</td>
</tr>
<tr>
<td>8,000</td>
<td>2.87</td>
</tr>
<tr>
<td>16,000</td>
<td>11.00</td>
</tr>
<tr>
<td>32,000</td>
<td>44.64</td>
</tr>
<tr>
<td>64,000</td>
<td>177.48</td>
</tr>
</tbody>
</table>

ThreeSum.java

ThreeSumDeluxe.java
Common order-of-growth hypotheses

**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 hypotheses

<table>
<thead>
<tr>
<th>growth rate</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>( a = b + c; )</td>
<td>statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td>while ((N &gt; 1)) { (N = N / 2; \ldots ) }</td>
<td>divide in half</td>
<td>binary search</td>
<td>( \sim 1 )</td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td>for (int (i = 0; i &lt; N; i++)) { \ldots }</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>( \sim 2 )</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>quadratic</td>
<td>for (int (i = 0; i &lt; N; i++)) |</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<pre><code>                 | for (int \(j = 0; j &lt; N; j++\)) \{ \ldots \} | double loop            | check all pairs  | 4                |
</code></pre>
<p>| ( N^3 )   | cubic          | for (int (i = 0; i &lt; N; i++)) |
| for (int (j = 0; j &lt; N; j++)) |
| for (int (k = 0; k &lt; N; k++)) { \ldots } | triple loop            | check all triples| 8                |
| ( 2^N )   | exponential    | [see combinatorial search lecture]            | exhaustive search      | check all possibilities | ( T(N) ) |</p>
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>description</th>
<th>effect on a program that runs for a few seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>independent of input size</td>
<td>time for 100x more data</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td>nearly independent of input size</td>
<td>time for 100x more data</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td>optimal for N inputs</td>
<td>effect on a program that runs for a few seconds</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>nearly optimal for N inputs</td>
<td>time for 100x more data</td>
</tr>
<tr>
<td>N²</td>
<td>quadratic</td>
<td>not practical for large problems</td>
<td>size for 100x faster computer</td>
</tr>
<tr>
<td>N³</td>
<td>cubic</td>
<td>not practical for medium problems</td>
<td>size for 100x faster computer</td>
</tr>
<tr>
<td>2ᴺ</td>
<td>exponential</td>
<td>useful only for tiny problems</td>
<td>time for 100x more data</td>
</tr>
</tbody>
</table>
• estimating running time
• mathematical analysis
• order-of-growth hypotheses
• input models
• measuring space
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 guarantee for all inputs.

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

Ex 1. Array accesses for brute-force 3-sum.
- Best: ~ $\frac{1}{2} N^3$
- Average: ~ $\frac{1}{6} N^3$
- Worst: ~ $\frac{1}{6} N^3$

Ex 2. Compares for insertion sort.
- Best (ascending order): ~ $N$.
- Average (random order): ~ $\frac{1}{4} N^2$
- Worst (descending order): ~ $\frac{1}{2} N^2$

(details in Lecture 4)
**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><strong>Tilde</strong></td>
<td>leading term</td>
<td>~ $10N^2$</td>
<td>$10N^2$ $10N^2 + 22N\log N$ $10N^2 + 2N + 37$</td>
<td>provide approximate model</td>
</tr>
<tr>
<td><strong>Big Theta</strong></td>
<td>asymptotic growth rate</td>
<td>$\Theta(N^2)$</td>
<td>$N^2$ $9000N^2$ $5N^2 + 22N\log N + 3N$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td><strong>Big Oh</strong></td>
<td>$\Theta(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$N^2$ $100N$ $22N\log N + 3N$</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td><strong>Big Omega</strong></td>
<td>$\Theta(N^2)$ and larger</td>
<td>$\Omega(N^2)$</td>
<td>$9000N^2$ $N^5$ $N^3 + 22N\log N + 3N$</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>

*Common mistake.* Interpreting big-Oh as an approximate model.
Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.
- Big-Oh notation suppresses leading constant.
- Big-Oh notation only provides upper bound (not lower bound).
› estimating running time
› mathematical analysis
› order-of-growth hypotheses
› input models
› measuring space
Typical memory requirements for primitive types in Java

**Bit.** 0 or 1.

**Byte.** 8 bits.

**Megabyte (MB).** 1 million bytes.

**Gigabyte (GB).** 1 billion bytes.

<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>
Typical memory requirements for arrays in Java

**Array overhead.** 16 bytes.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>2N + 16</td>
</tr>
<tr>
<td>int[]</td>
<td>4N + 16</td>
</tr>
<tr>
<td>double[]</td>
<td>8N + 16</td>
</tr>
</tbody>
</table>

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

**Ex.** An N-by-N array of doubles consumes ~ 8N^2 bytes of memory.
Typical memory requirements for objects in Java

**Object overhead.** 8 bytes.

**Reference.** 4 bytes.

**Ex 1.** A Complex object consumes 24 bytes of memory.

```java
public class Complex {
    private double re;
    private double im;
    ...
}
```

24 bytes

- 8 bytes overhead for object
- 8 bytes for the `re` value
- 8 bytes for the `im` value

24 bytes

- Reference
- 4 bytes for the reference
- 8 bytes overhead for each object
- Memory needed for the object's instance variables
Typical memory requirements for objects in Java

Object overhead. 8 bytes.
Reference. 4 bytes.

Ex 2. A virgin string of length N consumes ~ 2N bytes of memory.

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

8 bytes overhead for object

4 bytes
4 bytes
4 bytes
4 bytes for reference (plus 2N + 16 bytes for array)

2N + 40 bytes
Example 1

Q. How much memory does QuickUWPC use as a function of $N$?

A.

```java
public class QuickUWPC {
    private int[] id;
    private int[] sz;

    public QuickUWPC(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;
    }

    public boolean find(int p, int q) {
        // ... }

    public void unite(int p, int q) {
        // ... }
}
```
Example 2

Q. How much memory does this code fragment use as a function of $N$?
A.

```java
... 
int N = Integer.parseInt(args[0]);
for (int i = 0; i < N; i++) {
    int[] a = new int[N];
    ...
}
...
```

Remark. Java automatically reclaims memory when it is no longer in use.
Turning the crank: summary

In principle, accurate mathematical models are available. In practice, approximate mathematical models are easily achieved.

Timing may be flawed?
• Limits on experiments insignificant compared to other sciences.

• Mathematics might be difficult?
• Only a few functions seem to turn up.
• Doubling hypothesis cancels complicated constants.

Actual data might not match input model?
• Need to understand input to effectively process it.
• Approach 1: design for the worst case.
• Approach 2: randomize, depend on probabilistic guarantee.