### Analysis of Algorithms

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

**Updated from:**
Algorithms in Java, Chapter 2
Intro to Programming in Java, Section 4.1

---

### Reasons to analyze algorithms

- Predict performance.
- Compare algorithms.
- Provide guarantees.
- Understand theoretical basis.

This course (COS 226)

Theory of algorithms (COS 423)

---

### 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.

---

**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
**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*.

![Graph showing time vs. size for linear, linearithmic, and quadratic growth](image)

---

**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.**

---

**Experimental algorithmics**

*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.

*Why is my program so slow?*
Example: 3-sum

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

Application. Deeply related to problems in computational geometry.

```java
public class ThreeSum {
    public static int count(long[] 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) {
        int[] a = StdArrayIO.readLong1D();
        StdOut.println(count(a));
    }
}
```

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

Measuring the running time

```java
Stopwatch stopwatch = new Stopwatch();
ThreeSum.count(a);
double time = stopwatch.elapsedTime();
StdOut.println("Running time: "+time+" seconds");
```

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

Data analysis. Observe and plot running time as a function of input size $N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds) $\dagger$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.26</td>
</tr>
<tr>
<td>2048</td>
<td>2.16</td>
</tr>
<tr>
<td>4096</td>
<td>17.18</td>
</tr>
<tr>
<td>8192</td>
<td>137.76</td>
</tr>
</tbody>
</table>

$\dagger$ Running Linux on Sun-Fire-X4100

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

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

Hypothesis. Running time grows cubically with input size: $c N^3$.

Empirical analysis

Prediction and verification

Hypothesis. $2.5 \times 10^{-10} \times N^3$ seconds for input of size $N$.

Prediction. 17.18 seconds for $N = 4,096$.

Observations.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>17.18</td>
</tr>
<tr>
<td>4096</td>
<td>17.15</td>
</tr>
<tr>
<td>4096</td>
<td>17.17</td>
</tr>
</tbody>
</table>

Prediction. 1100 seconds for $N = 16,384$.

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>

Doubling hypothesis

Q. What is effect on the running time of doubling the size of the input?

Regression. Fit straight line through data points: $c N^a$; slope $= 3$.

Hypothesis. Running time grows cubically with input size: $c N^3$.

Observations.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds) $\dagger$</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.03</td>
<td>-</td>
</tr>
<tr>
<td>1024</td>
<td>0.26</td>
<td>7.88</td>
</tr>
<tr>
<td>2048</td>
<td>2.16</td>
<td>8.43</td>
</tr>
<tr>
<td>4096</td>
<td>17.18</td>
<td>7.96</td>
</tr>
<tr>
<td>8192</td>
<td>137.76</td>
<td>7.96</td>
</tr>
</tbody>
</table>

Bottom line. Quick way to formulate a power law hypothesis.

Doubling hypothesis

Q. What is effect on the running time of doubling the size of the input?

Regression. Fit straight line through data points: $c N^a$; slope $= 3$.

Hypothesis. Running time grows cubically with input size: $c N^3$.

Observations.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds) $\dagger$</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.03</td>
<td>-</td>
</tr>
<tr>
<td>1024</td>
<td>0.26</td>
<td>7.88</td>
</tr>
<tr>
<td>2048</td>
<td>2.16</td>
<td>8.43</td>
</tr>
<tr>
<td>4096</td>
<td>17.18</td>
<td>7.96</td>
</tr>
<tr>
<td>8192</td>
<td>137.76</td>
<td>7.96</td>
</tr>
</tbody>
</table>

Bottom line. Quick way to formulate a power law 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

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.

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>
</tbody>
</table>

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

Donald Knuth
1974 Turing Award
### Cost of basic operations

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds</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?

```
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>$N + 2$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>less than comparison</td>
<td>1/2 (N + 1) (N + 2)</td>
</tr>
<tr>
<td>equal to comparison</td>
<td>1/2 N (N − 1)</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N − 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$\leq 2N$</td>
</tr>
</tbody>
</table>

Betweeen $N$ (no zeros) and $2N$ (all zeros)

### Example: 2-sum

Q. How many instructions as a function of 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++;
```

<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 comparison</td>
<td>1/2 (N + 1) (N + 2)</td>
</tr>
<tr>
<td>equal to comparison</td>
<td>1/2 N (N − 1)</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N − 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$\leq 2N$</td>
</tr>
</tbody>
</table>

$0 + 1 + 2 + \ldots + (N − 1) = \frac{1}{2} N^2 N − 1 = \frac{1}{2} N(N − 1)$

Exercise 1. $6 N^3 + 20 N + 16 \sim 6 N^3$
Exercise 2. $6 N^3 + 100 N^{43} + 56 \sim 6 N^3$
Exercise 3. $6 N^3 + 17 N^2 \log N + 7 N \sim 6 N^3$

• 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

### Tilde notation

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

discard lower-order terms (e.g., $N = 1000$ 6 trillion vs. 169 million)
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++;
```

```
operation | frequency | cost | total cost
--- | --- | --- | ---
variable declaration | ~ N | c1 | ~ c1 N
assignment statement | ~ N | c2 | ~ c2 N
less than comparison | ~ 1/2 N^2 | c3 | ~ c3 N^2
equal to comparison | ~ 1/2 N^2 | c3 | ~ c3 N^2
array access | ~ N^2 | c4 | ~ c4 N^2
increment | ≤ N^2 | c5 | ≤ c5 N^2
```

```
total ~ c N^2
```

Example: 3-sum

Q. How many instructions as a function of N?

```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] + a[k] == 0) cnt++;
    return cnt;
}
```

```
( \frac{N}{6} ) = \frac{N(N-1)(N-2)}{6} = \frac{1}{6} N^3 
```

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

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.

Example models:

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

Bottom line. We use approximate models in this course: \( T_N \sim c N^3 \).
To determine order-of-growth:
• Assume a power law \( T_N \sim c N^a \).
• Estimate exponent \( a \) with doubling hypothesis.
• Validate with mathematical analysis.

**Ex. ThreeSumDeluxe.java**

Food for thought. How is it implemented?

**Caveat.** Can’t identify logarithmic factors with doubling hypothesis.

---

### Practical implications of order-of-growth

**Q.** How long to process millions of inputs?

**Ex.** Population of NYC was "millions" in 1970s; still is.

**Q.** How many inputs can be processed in minutes?

**Ex.** Customers lost patience waiting "minutes" in 1970s; still do.

For back-of-envelope calculations, assume:

<table>
<thead>
<tr>
<th>decade</th>
<th>processor speed</th>
<th>instructions per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970s</td>
<td>1 MHz</td>
<td>( 10^6 )</td>
</tr>
<tr>
<td>1980s</td>
<td>10 MHz</td>
<td>( 10^7 )</td>
</tr>
<tr>
<td>1990s</td>
<td>100 MHz</td>
<td>( 10^8 )</td>
</tr>
<tr>
<td>2000s</td>
<td>1 GHz</td>
<td>( 10^9 )</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>age of universe</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

---

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( \sim 1 )</td>
<td>constant</td>
<td>( a = b + c; )</td>
<td>statement</td>
<td>add two numbers</td>
</tr>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td>( \text{while } (N &gt; 1) ) ( { \text{...} } )</td>
<td>divide in half</td>
<td>binary search</td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td>( \text{for (int } i = 0; i &lt; N; i++) } ( { \text{...} } )</td>
<td>loop</td>
<td>find the maximum</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>linearithmic</td>
<td>{ see lecture 5 }</td>
<td>divide and conquer</td>
<td>mergesort</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>quadratic</td>
<td>( \text{for (int } i = 0; i &lt; N; i++) } ( { \text{...} } )</td>
<td>double loop</td>
<td>check all pairs</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>cubic</td>
<td>( \text{for (int } i = 0; i &lt; N; i++) } ( { \text{...} } )</td>
<td>triple loop</td>
<td>check all triples</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>exponential</td>
<td>{ see lecture 24 }</td>
<td>exhaustive search</td>
<td>check all possibilities</td>
</tr>
</tbody>
</table>

---

### Good news.

The small set of functions

- \( 1 \)
- \( \log N \)
- \( N \)
- \( N \log N \)
- \( N^2 \)
- \( 2^N \)

suffices to describe order-of-growth of typical algorithms.
### Practical implications of order-of-growth

#### Growth rate

<table>
<thead>
<tr>
<th>Growth Rate</th>
<th>Problem Size Solvable in Minutes</th>
<th>Time to Process Millions of Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any</td>
<td>instant</td>
</tr>
<tr>
<td>( \log N )</td>
<td>any</td>
<td>instant</td>
</tr>
<tr>
<td>( N )</td>
<td>millions</td>
<td>tens of minutes</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>hundreds of thousands</td>
<td>millions</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>hundreds</td>
<td>thousand</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>hundred</td>
<td>thousand</td>
</tr>
</tbody>
</table>

<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>-</td>
</tr>
<tr>
<td>( \log N )</td>
<td>Logarithmic</td>
<td>nearly independent of input size</td>
<td>-</td>
</tr>
<tr>
<td>( N )</td>
<td>Linear</td>
<td>optimal for ( N ) inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>Linearithmic</td>
<td>nearly optimal for ( N ) inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>Quadratic</td>
<td>not practical for large problems</td>
<td>several hours</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>Cubic</td>
<td>not practical for medium problems</td>
<td>several weeks</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>Exponential</td>
<td>useful only for tiny problems</td>
<td>forever</td>
</tr>
</tbody>
</table>

### Types of analyses

- **Best case.** Running time determined by easiest inputs.  
  **Ex.** \( N-1 \) compares to insertion sort \( N \) elements in ascending order.

- **Worst case.** Running time guarantee for all inputs.  
  **Ex.** No more than \( \frac{1}{2} N^2 \) compares to insertion sort any \( N \) elements.

- **Average case.** Expected running time for "random" input.  
  **Ex.** \( \sim \frac{1}{2} N^2 \) compares on average to insertion sort \( N \) random elements.
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>( N^2 ) ( 9000 \ N^2 ) ( 5 \ N^2 + 22 \ N \log N + 3N )</td>
<td>classify algorithms</td>
</tr>
<tr>
<td>Big Oh</td>
<td>( \Theta(N^2) ) and smaller</td>
<td>( O(N^2) )</td>
<td>( 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>( 9000 \ N^2 ) ( N^3 ) ( N^2 + 22 \ N \log N + 3 N )</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>

Ex 1. Our brute-force 3-sum algorithm takes \( \Theta(N^3) \) time.

Ex 2. Conjecture: worst-case running time for any 3-sum algorithm is \( \Omega(N^2) \).

Ex 3. Insertion sort uses \( O(N^2) \) compares to sort any array of \( N \) elements; it uses \( \sim N \) compares in best case (already sorted) and \( \sim \frac{1}{2} N^2 \) compares in the worst case (reverse sorted).

Ex 4. The worst-case height of a tree created with union find with path compression is \( \Theta(N) \).

Ex 5. The height of a tree created with weighted quick union is \( O(\log N) \).

Predictions and guarantees

**Theory of algorithms.** Worst-case running time of an algorithm is \( O(f(N)) \).

**Advantages**
- describes guaranteed performance.
- \( O \)-notation absorbs input model.

**Challenges**
- Cannot use to predict performance.
- Cannot use to compare algorithms.

Experimental algorithmics. Given input model, average-case running time is \( \sim c f(N) \).

**Advantages.**
- Can use to predict performance.
- Can use to compare algorithms.

**Challenges.**
- Need to develop accurate input model.
- May not provide guarantees.
Typical memory requirements for primitive types in Java

<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 on a typical machine.

<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>

**Object overhead.** 8 bytes on a typical machine.

**Reference.** 4 bytes on a typical machine.

**Ex 1.** Each complex object consumes 24 bytes of memory.

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

Q. What’s the biggest double[] array you can store on your computer?

A typical computer in 2008 has about 16GB memory.

```
public class Document {
   private int count;
   private int offset;
   private char[] val;
   ...
}
```

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

Typical memory requirements for objects in Java

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

8 bytes overhead for object

8 bytes

8 bytes

24 bytes

24 bytes

Object overhead

24 bytes

double

object overhead

re

im
Typical memory requirements for objects in Java

Object overhead.  8 bytes on a typical machine.
Reference.  4 bytes on a typical machine.

Ex 2.  A string of length $N$ consumes $2N + 40$ bytes.

```
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 this program use as a function of $N$?

```
public class RandomWalk {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      int[] count = new int[N][N];
      int x = N/2;
      int y = N/2;
      for (int i = 0; i < N; i++) {
         // no new variable declared in loop
         ...
         count[x][y]++;
      }
   }
}
```

Example 2

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

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