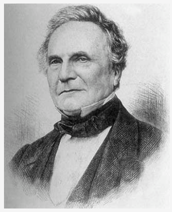
Analysis of Algorithms

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

Reference:
Algorithms in Java, Chapter 2
Intro to Programming in Java, Section 4.1
`http://www.cs.princeton.edu/algs4`

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*



Charles Babbage (1864)

how many times do you have to turn the crank?

Analytic Engine

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

this course (COS 226)

Provide guarantees.

Understand theoretical basis.

theory of algorithms (COS 423)

client gets poor performance because programmer did not understand performance characteristics

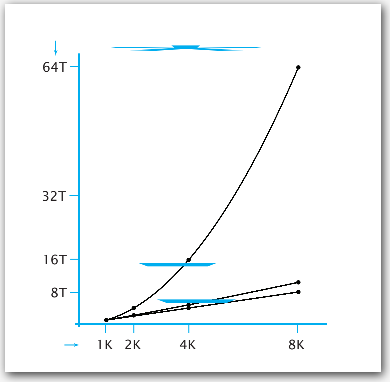
Primary practical reason: avoid performance bugs.

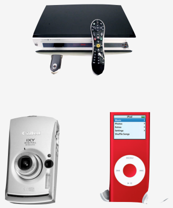
Some algorithmic successes

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.

Freidrich Gauss
1805



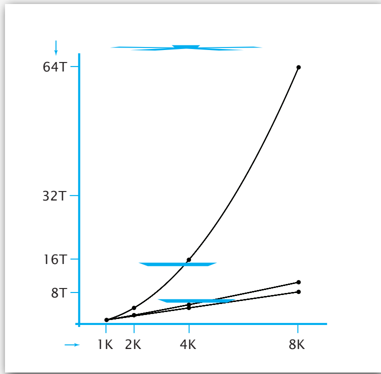


Some algorithmic successes

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.

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.

Example: 3-sum

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

```
% 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.

3-sum: brute-force algorithm

```
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)
   {
      long[] a = StdArrayIO.readLong1D();
      StdOut.println(count(a));
   }
}
```

check each triple

Empirical analysis

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

N	time (seconds) †
1024	0.26
2048	2.16
4096	17.18
8192	137.76

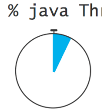
† Running Linux on Sun-Fire-X4100

Measuring the running time

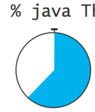
Q. How to time a program?

A. Manual.

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




```
0
% java ThreeSum < 2Kints.txt
```




```
2
391930676 -763182495 371251819
-326747290 802431422 -475684132
```

Measuring the running time

Q. How to time a program?

A. Automatic.

```
Stopwatch stopwatch = new Stopwatch();

ThreeSum.count(a);

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

client code

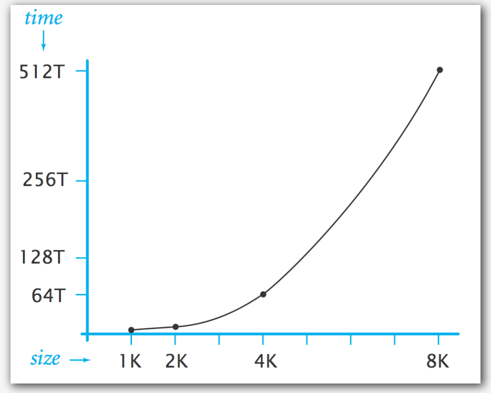
```
public class Stopwatch
{
   private final long start = System.currentTimeMillis();

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

implementation (part of stdlib.jar)

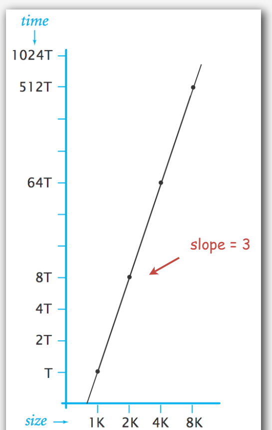
Data analysis

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



Data analysis

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



power law

Regression. Fit straight line through data points: $a\,N^{\,b}$.

slope

Hypothesis. Running time grows cubically with 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.

N	time (seconds) †	ratio	lg ratio
512	0.03	-	
1024	0.26	7.88	2.98
2048	2.16	8.43	3.08
4096	17.18	7.96	2.99
8192	137.76	7.96	2.99

seems to converge to a constant b ≈ 3

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

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!

N	time (seconds)
4096	17.18
4096	17.15
4096	17.17

$17.17 = a \times 4096^3$
$\Rightarrow a = 2.5 \times 10^{-10}$

Refined hypothesis. Running time is about $2.5 \times 10^{-10} \times N^3$ seconds.

Prediction. $1{,}100$ seconds for $N = 16{,}384$.
Observation.

N	time (seconds)
16384	1118.86

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

Imprecise measurements

Statistics. Use statistics to ensure confidence in timing data.

```
public class ThreeSumStats {

    public static void main(String[] args)  {
        int N = Integer.parseInt(args[0]);    // problem size
        int T = Integer.parseInt(args[1]);    // repeat T times
        double[] time = new double[T];        // time for tth trial
        long[] a = new long[N];

        for (int t = 0; t < T; t++) {
            for (int i = 0; i < N; i++)
                a[i] = -500000 + StdRandom.uniform(1000000);
            Stopwatch timer = new Stopwatch();
            int count = ThreeSum.count(a);
            time[t] = timer.elapsedTime();
        }

        System.out.println("mean   = " + StdStats.mean(time));
        System.out.println("stddev = " + StdStats.stddev(time));
    }
}
```

Caveat. No confidence when stddev is bigger than mean!

War story (from COS 126)

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

```
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] = ...
    ...
}
```

Jenny. ~ $c_1 N^2$ seconds.

Kenny. ~ $c_2 N$ seconds.

N	time
1024	0.11
2048	0.35
4096	1.6
9182	6.5

Jenny

N	time
256	0.5
512	1.1
1024	1.9
2048	3.9

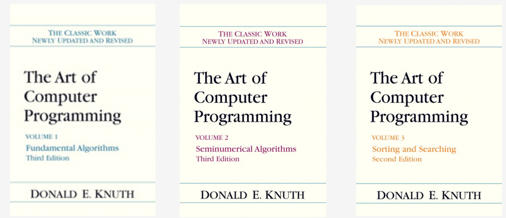
Kenny

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

Mathematical models for running time

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

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating point add	`a + b`	4.6
floating point multiply	`a * b`	4.2
floating point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

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

Cost of basic operations

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

Novice mistake. Abusive string concatenation.

Example: 1-sum

Q. How many instructions as a function of N?

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

operation	frequency
variable declaration	2
assignment statement	2
less than comparison	$N+1$
equal to comparison	N
array access	N
increment	$\leq 2\,N$

between 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++;
```

operation	frequency
variable declaration	$N+2$
assignment statement	$N+2$
less than comparison	$1/2\,(N+1)\,(N+2)$
equal to comparison	$1/2\,N\,(N-1)$
array access	$N\,(N-1)$
increment	$\leq N^2$

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

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. $6\,N^3 + 20\,N + 16 \sim 6\,N^3$

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

Ex 3. $6\,N^3 + \underbrace{17\,N^2 \lg N + 7\,N}_{} \sim 6\,N^3$

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

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

Example: 2-sum

Q. How long will it take 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++;
```

"inner loop"

operation	frequency	time per op	total time
variable declaration	$\sim N$	c_1	$\sim c_1 N$
assignment statement	$\sim N$	c_2	$\sim c_2 N$
less than comparison	$\sim 1/2\,N^2$	c_3	$\sim c_3 N^2$
equal to comparison	$\sim 1/2\,N^2$		
array access	$\sim N^2$	c_4	$\sim c_4 N^2$
increment	$\leq N^2$	c_5	$\leq c_5 N^2$
total			$\sim c\,N^2$

Example: 3-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++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

"inner loop"

~ 1

$\sim N$

$\sim N^2 / 2$

$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$

depends on input data

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.

costs (depend on machine, compiler)

$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$

A = variable declarations

B = assignment statements

C = compare

D = array access

E = increment

frequencies (depend on algorithm, input)

Bottom line. We use approximate models in this course: $T_N \sim c \; N^3$.

- 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?

N	time (seconds) †
1,000	0.43
2,000	0.53
4,000	1.01
8,000	2.87
16,000	11.00
32,000	44.64
64,000	177.48

observations

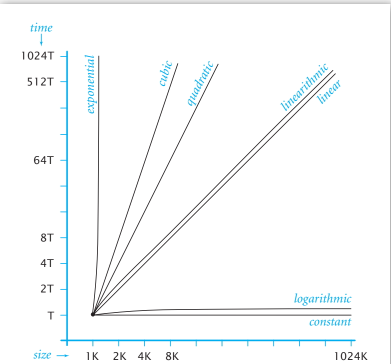
Caveat. Can't identify logarithmic factors with doubling hypothesis.

Common order-of-growth hypotheses

Good news. the small set of functions

$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3,$ and 2^N

suffices to describe order-of-growth of typical algorithms.



Orders of growth (log-log plot)

growth rate	name	T(2N) / T(N)
1	constant	1
log N	logarithmic	~ 1
N	linear	2
N log N	linearithmic	~ 2
N^2	quadratic	4
N^3	cubic	8
2^N	exponential	T(N)

factor for doubling hypothesis

Common order-of-growth hypotheses

growth rate	name	typical code framework	description	example
1	constant	`a = b + c;`	statement	add two numbers
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum
N log N	linearithmic	[see lecture 5]	divide and conquer	mergesort
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs
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
2^N	exponential	[see lecture 24]	exhaustive search	check all possibilities

Practical implications of order-of-growth

Q. How many inputs can be processed in minutes?

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

Q. How long to process millions of inputs?

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

For back-of-envelope calculations, assume:

decade	processor speed	instructions per second
1970s	1 MHz	10^6
1980s	10 MHz	10^7
1990s	100 MHz	10^8
2000s	1 GHz	10^9

seconds	equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
…	forever
10^{17}	age of universe

Practical implications of order-of-growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
N log N	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millennia

Practical implications of order-of-growth

growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	-	-
log N	logarithmic	nearly independent of input size	-	-
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4-5x
2^N	exponential	useful only for tiny problems	forever	1x

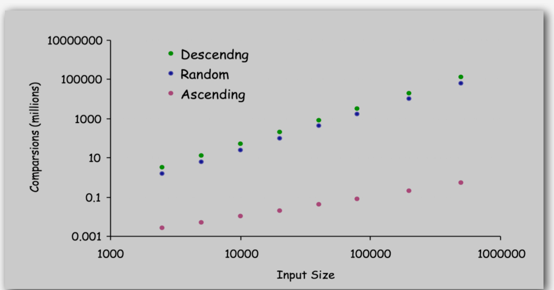
- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- **input models**
- measuring space

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. $\sim \frac{1}{4} N^2$ compares on average to insertion sort N random elements.



Commonly-used notations

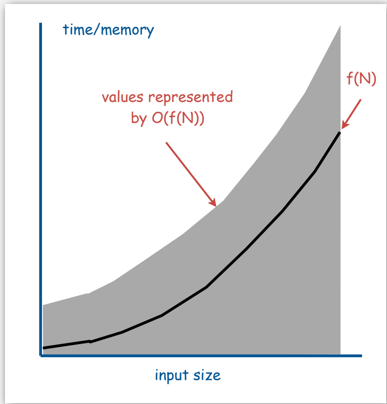
notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\,N^2$	$10\,N^2$ $10\,N^2 + 22\,N \log N$ $10\,N^2 + 2\,N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	N^2 $9000\,N^2$ $5\,N^2 + 22\,N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	N^2 $100\,N$ $22\,N \log N + 3\,N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$9000\,N^2$ N^5 $N^3 + 22\,N \log N + 3\,N$	develop lower bounds

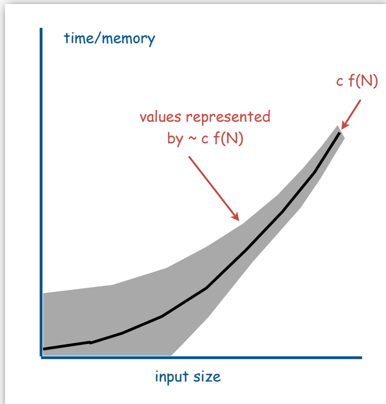
Common mistake. Using big-Oh when you mean big-Omega.

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).





Examples

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 the best case (already sorted) and $\sim \frac{1}{2}N^2$ compares in the worst case (reverse sorted).

1

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 $O(\log N)$.

base of logarithm absorbed by big-Oh

$$\log_a N = \frac{1}{\log_b a} \log_b N$$

- 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). 2^{20} bytes ~ 1 million bytes.
Gigabyte (GB). 2^{30} bytes ~ 1 billion bytes.

type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

Typical memory requirements for arrays in Java

Array overhead. 16 bytes.

type	bytes
`char[]`	2N + 16
`int[]`	4N + 16
`double[]`	8N + 16

one-dimensional arrays

type	bytes
`char[][]`	$2N^2$ + 20N + 16
`int[][]`	$4N^2$ + 20N + 16
`double[][]`	$8N^2$ + 20N + 16

two-dimensional arrays

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

A.

typical computer in 2008 has about 2GB 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.

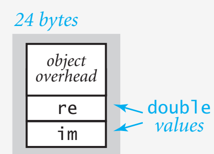
```
public class Complex
{
   private double re;
   private double im;
   ...
}
```

8 bytes overhead for object

8 bytes

8 bytes

24 bytes



Typical memory requirements for objects in Java

Object overhead. 8 bytes.

Reference. 4 bytes.

Ex 2. A virgin `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

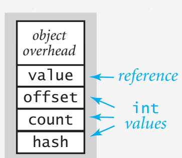
4 bytes

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 data type use as a function of N?

A.

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

   public QuickUnion(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.

```
...
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.

Out of memory

Q. What if I run out of memory?

```
% java RandomWalk 10000
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

% java -Xmx 500m RandomWalk 10000
...

% java RandomWalk 30000
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

% java -Xmx 4500m RandomWalk 30000
Invalid maximum heap size: -Xmx4500m
The specified size exceeds the maximum representable size.
Could not create the Java virtual machine.
```


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.