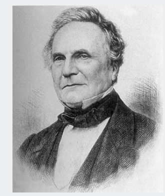
Analysis of Algorithms

- overview
- experiments
- models
- case study
- hypotheses

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

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)

Analytic Engine

how many times do you have to turn the crank?

Reasons to analyze algorithms

Predict performance

Compare algorithms

Provide guarantees

Understand theoretical basis

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs

Client gets poor performance because programmer did not understand performance characteristics

Overview

Scientific analysis of algorithms:
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.

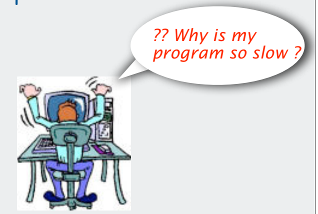
Principles.
- Experiments must be reproducible.
- Hypotheses must be falsifiable.

Universe = computer itself.

- overview
- **experiments**
- models
- case study
- hypotheses

Experimental algorithmics

Every time you run a program you are doing an experiment!



First step:
Debug your program!

Second step:
Decide on model for experiments on large inputs.

Third step:
Run the program for problems of increasing size.

Experimental evidence: measuring time

- Manual: 
- Automatic: `Stopwatch.java`

client code

```
Stopwatch sw = new Stopwatch();
// Run algorithm
double time = sw.elapsedTime();
StdOut.println("Running time: " + time + " seconds");
```

implementation

```
public class Stopwatch
{
   private final long start;

   public Stopwatch()
   {  start = System.currentTimeMillis();  }

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


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: we can run a huge number of experiments [stay tuned]

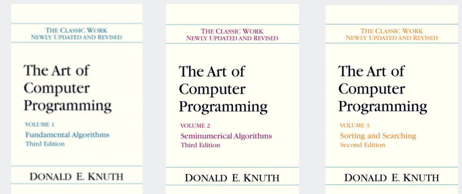
Approach 1: Settle for affordable approximate results
Approach 2: Count abstract operations (machine independent)

- overview
- experiments
- **models**
- case study
- hypotheses

Models for the analysis of algorithms

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

Developing models for algorithm performance

In principle, accurate mathematical models are available [Knuth]

In practice,

- formulas can be complicated
- advanced mathematics might be required

Ex.

costs (depend on machine, compiler)

$T_N = 24\,A_N + 11B_N + 4C_N + 3D_N + 7N + 9S_N$

where

$A_N = 2(N+1) / 3$

$B_N = (N + 1)\,(2H_{N+1} - 2H_3 - 1)/6 + 1/2$

$C_N = (N + 1)\,(2H_{N+1} - 2H_3 + 1)$

$D_N = (N + 1)(1 - 2H_3/3)$

$S_N = (N + 1)/5 - 1$

frequencies
(depend on algorithm, input)

Exact models best left for experts

Bottom line: We use approximate models in this course: $T_N \sim c\ N \log N$

all constants rolled into one

Commonly used notations to model running time

notation	provides	example	shorthand for	used to
Big Theta	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 $100\ N$ $22\ N \log N + 3N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$9000\ N^2$ N^5 $N^3 + 22\ N \log N + 3N$	develop lower bounds
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

used in this course

Predictions and guarantees

Theory of algorithms: The running time of an algorithm is O(f(N))

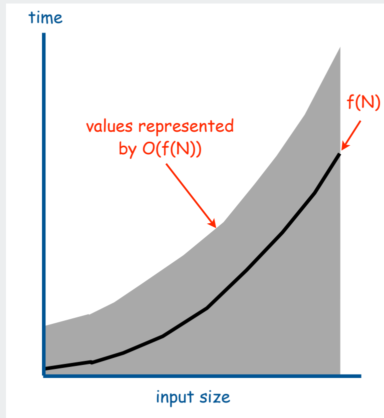
worst case implied

advantages

- describes guaranteed performance
- O-notation absorbs input model

challenges

- cannot use to predict performance
- cannot use to compare algorithms



Predictions and guarantees (continued)

This course: The running time of an algorithm is ~ c f(N)

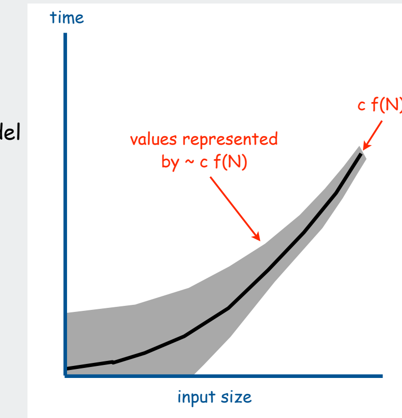
understanding of alg's dependence on input implied

advantages

- can use to predict performance
- can use to compare algorithms

challenges

- need to develop accurate input model
- may not provide guarantees



Case study [stay tuned for numerous algorithms and applications]

Sorting problem: rearrange N given items into ascending order

...	...
Hauser	Haskell
Hong	Hauser
Hsu	Hayes
Hayes	Hong
Haskell	Hornet
Hornet	Hsu
...	...

Basic operations: compares and exchanges

compare

```
public static void less(double x, double y)
{  return x < y; }
```

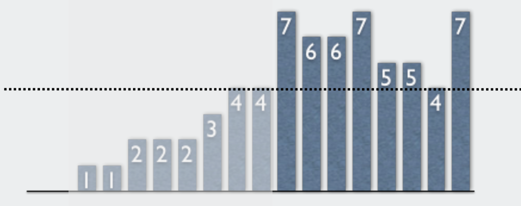
exchange

```
public static void exch(double[] a, int i, int j)
{
   double t = a[i];
   a[i] = a[j];
   a[j] = t;
}
```

Selection sort: an elementary sorting algorithm

Algorithm invariants

- ↑ scans from left to right.
- Elements to the left of ↑ are fixed and in ascending order.
- No element to left of ↑ is larger than any element to its right.



Selection sort inner loop

- move the pointer to the right

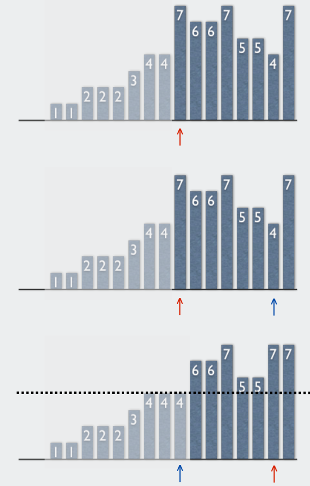
```
i++;
```

- identify index of minimum item on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```



Maintains algorithm invariants

Selection sort: Java implementation

```
public static void sort(double[] a)
{
   for (int i = 0; i < a.length; i++)
   {
      int min = i;
      for (int j = i+1; j < a.length; j++)
         if (less(a[j], a[min]))
            min = j;
      exch(a, i, min);
   }
}
```

most frequent operation ("inner loop")

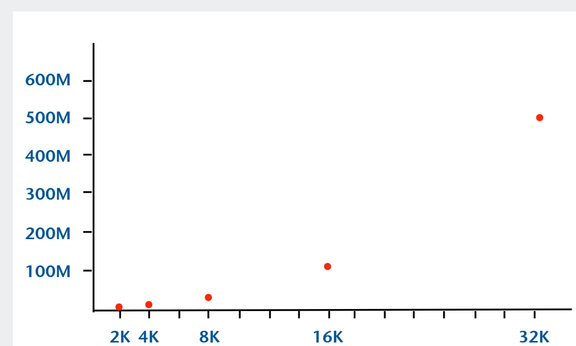
Selection sort: initial observations

Observe, tabulate and plot operation counts for various values of N.

- study most frequently performed operation (compares)
- input model: N random numbers between 0 and 1

add counter to `less()`

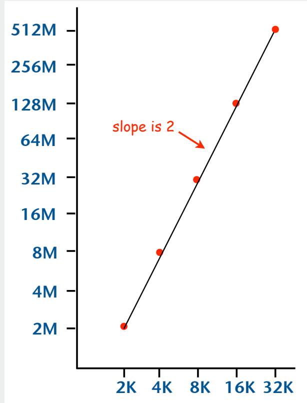
N	compares
2,000	2.1 million
4,000	7.9 million
8,000	32.1 million
16,000	125.9 million
32,000	514.7 million



Selection sort: experimental hypothesis

Data analysis. Plot # compares vs. input size on log-log scale.

N	compares
2,000	2.1 million
4,000	7.9 million
8,000	32.1 million
16,000	125.9 million
32,000	514.7 million



normal scale

$C = a N^b$

log-log scale

$\lg C = \lg a + b \lg N$

power law

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

slope

Hypothesis. # compares is $\sim N^2/2$

Selection sort: theoretical model

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

a[i]

each black entry is 1 compare

circled entry is min value found

gray entries are untouched

Hypothesis: number of compares is $N + (N-1) + \ldots + 2 + 1 \sim N^2/2$

$= N(N + 1) / 2$

$= N^2/2 + N/2$

$\sim N^2/2$

Selection sort: Prediction and verification

Hypothesis (experimental and theoretical). # compares is $\sim N^2/2$.

Prediction. 800 million compares for N = 40,000.

Observations.

N	compares
40,000	801.3 million
40,000	799.7 million
40,000	801.6 million
40,000	800.8 million

Verifies.

Prediction. 20 billion compares for N = 200,000.

Observation.

N	compares
200,000	19.997 billion

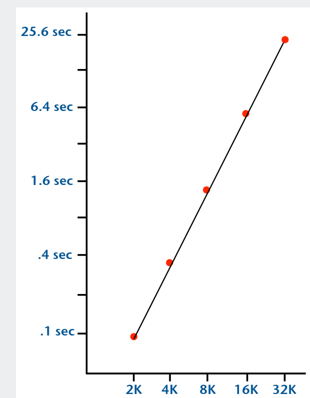
Verifies.

Selection sort: validation

Implicit assumptions
- constant cost per compare
- cost of compares dominates running time

Hypothesis: Running time is $\sim c N^2$
Validation: Observe actual running time.

N	observed time	$.23 \times 10^{-7} N^2$
2,000	0.1 seconds	0.1
4,000	0.4 seconds	0.4
8,000	1.5 seconds	1.5
16,000	5.6 seconds	5.9
32,000	23.2 seconds	23.5



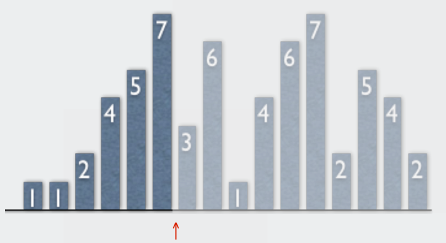
Regression fit validates hypothesis.

A scientific connection between program and natural world.

Insertion sort: another elementary sorting algorithm

Algorithm invariants

- ↑ scans from left to right.
- Elements to the left of ↑ are in ascending order.



Insertion sort inner loop

- move the pointer to the right

```
i++;
```

- moving from right to left, exchange a[i] with each larger element to its left

```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
        exch(a, j, j-1);
   else break;
```

Maintains algorithm invariants

Insertion sort: Java implementation

```
public static void sort(Comparable[] a)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0; j--)
         if (less(a[j], a[j-1]))
              exch(a, j, j-1);
         else break;
}
```


Insertion sort: theoretical model

a[i]

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

circled entry is inserted item

gray entries are untouched

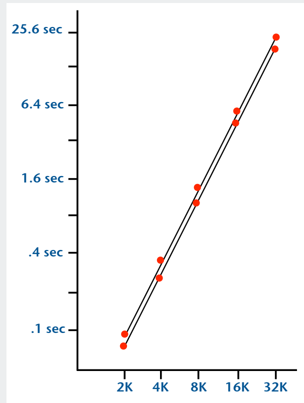
each black entry is 1 compare/exch

insertions are halfway back, on the average

Hypothesis: number of compares is (1 + 2 + ... + (N-1) + N)/2 ~ $N^2/4$ on the average, for randomly ordered input

Experimental comparison of insertion sort and selection sort

Plot both running times on log log scale
- slopes are the same (both 2)
- both are quadratic



Compute ratio of running times

```
% java SortCompare Insertion Selection 4000
For 4000 random double values
Insertion is 1.7 times faster than selection
```

Need detailed analysis
to prefer one over the other

Neither is useful for huge randomly-ordered files

Would Be Nice (if analysis of algorithms were always this easy), But

Mathematics might be difficult
Ex. It is known that properties of singularities of functions
in the complex plane play a role in analysis of many algorithms

Leading term might not be good enough
Ex. Selection sort could be linear-time if cost of exchanges is huge
↑
assumption that compares dominate may be invalid

Actual data might not match model
Ex. Insertion sort could be linear-time if keys are roughly in order
↑
assumption that input is randomly ordered may be invalid

Timing may be flawed
- different results on different computers
- different results on same computer at different times

Practical approach to developing hypotheses

First step: determine asymptotic growth rate for chosen model
- approach 1: run experiments, regression
- approach 2: do the math
- best: do both

Good news: the relatively small set of functions
1, $\log N$, N, $N \log N$, N^2, N^3, and 2^N
suffices to describe asymptotic growth rate of typical algorithms

After determining growth rate
- use doubling hypothesis (to predict performance)
- use ratio hypothesis (to compare algorithms)

Common asymptotic-growth hypotheses (summary)

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 next lecture]	divide and conquer	sort an array
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

Aside: practical implications of asymptotic growth

For back-of-envelope calculations, assume

decade	processor speed	instructions per second
1970s	1M Hz	10^6
1980s	10M Hz	10^7
1990s	100M Hz	10^8
2000s	1G Hz	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

How long to process millions of inputs?

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

How many inputs can be processed in minutes?

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

Aside: practical implications of asymptotic 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	millenia

Practical implications of asymptotic-growth: another view

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	a few seconds	same
log N	logarithmic	nearly independent of input size	a few seconds	same
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 large problems	several weeks	4-5x
2^N	exponential	useful only for tiny problems	forever	1x

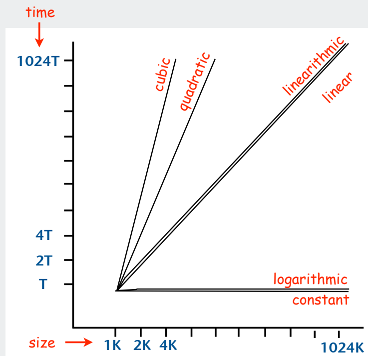
Developing asymptotic order of growth hypotheses with doubling

To formulate hypothesis for asymptotic growth rate:

- compute T(2N)/T(N) as accurately (and for N as large) as is affordable
- use this table

ratio	hypothesis	reason
1	constant or logarithmic	c / c = 1 c log 2N / c log N ~ 1
2	linear or linearithmic	c 2N / c N = 2 c 2 N log (2N) / c N log N ~ 2
4	quadratic	c $(2N)^2$ / c N^2 = 4
9	cubic	c $(2N)^2$ / c N^2 = 9

= 2 log(2N)/log N
= 2 (log 2 + log N)/log N
= 2 + 2 log 2/log N
~ 2



Example revisited: methods for timing sort algorithms

Compute time to sort `a[]` with `alg`

```
public static double time(String alg, Double[] a)
{
   Stopwatch sw = new Stopwatch();
   if (alg.equals("Insertion")) Insertion.sort(a);
   if (alg.equals("Selection")) Selection.sort(a);
   if (alg.equals("Shell")) Shell.sort(a);
   if (alg.equals("Merge")) Merge.sort(a);
   if (alg.equals("Quick")) Quick.sort(a);
   return sw.elapsedTime();
}
```

Compute total time to to sort `trials` arrays of N random doubles with `alg`

```
public static double timetrials(String alg, int N, int trials)
{
   double total = 0.0;
   Double[] a = new Double[N];
   for (int t = 0; t < trials; t++)
   {
      for (int i = 0; i < N; i++)
         a[i] = StdRandom.uniform();
      total += time(alg, a);
   }
   return total;
}
```


Developing asymptotic order of growth hypotheses with doubling

CAUTION

THIS CODE MAY NOT BE READY FOR THE REAL WORLD

```
public class SortGrowth
{
   public static void main(String[] args)
   {
      String alg = args[0];
      int N = 1000;
      if (args.length > 1)
           N = Integer.parseInt(args[1]);
      int trials = 100;
      if (args.length > 2)
           trials = Integer.parseInt(args[2]);
      double ratio = timetrials(alg, 2*N, trials);
                              / timetrials(alg, N, trials);
      StdOut.printf("Ratio is %f\n", ratio);
      if (ratio > 1.8 && ratio < 2.2)
         StdOut.printf("   %s is linear or linearithmic\n", alg);
      if (ratio > 3.8 && ratio < 4.2)
         StdOut.printf("   %s is quadratic\n", alg);

   }
}
```

```
% java SortGrowth Selection
Ratio is 4.1
  Selection is quadratic
```

```
% java SortGrowth Insertion
Ratio is 3.645756

% java SortGrowth Insertion 4000 1000
Ratio is 3.969934
  Insertion is quadratic
```


Predicting performance with doubling hypotheses

A practical approach to predict running time:

- analyze algorithm and run experiments to develop hypothesis that asymptotic growth rate of running time is ~ c T(N)
- run algorithm for some value of N, measure running time
- prediction: increasing input size by a factor of 2
  increases running time by a factor of T(2N)/T(N)

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	9

Example: selection sort

N	observed time
2,000	0.1 seconds
4,000	0.4 seconds
8,000	1.5 seconds
16,000	5.6 seconds
32,000	23.2 seconds

numbers increase by a factor of 2

numbers increase by a factor of 4

Use algorithm itself to implicitly compute leading-term constant

Predicting performance with doubling hypotheses

CAUTION
THIS CODE MAY NOT BE READY FOR THE REAL WORLD

```
public class SortPredict
{
   public static void main(String[] args)
   {
      String alg = args[0];
      int trials = 100;
      if (args.length > 1) trials = Integer.parseInt(args[1]);
      StdOut.printf("Seconds for %d trials\n", trials);
      StdOut.printf("       predicted actual\n  1000        ");
      double old = Double.POSITIVE_INFINITY;
      for (int N = 1000; true; N = 2*N)
      {
         total = timeTrials(alg, N, trials);
         double guess = (total/old)*total;
         StdOut.printf(" %7.1f\n %5d %7.1f", total, 2*N, guess);
         old = total;
      }
   }
}
```

```
% java SortPredict Selection
Seconds for 100 trials
          predicted   actual
  1000                  0.9
  2000         0.0      3.5
  4000        13.9     14.4
  8000        58.8     58.9
 16000       240.9    239.2
 32000       971.6
```

Note: `SortGrowth` is not needed!
[This code works for any power law.]

and deep math says that running time of typical algs must satisfy power law

Comparing algorithms with ratio hypotheses

A practical way to compare algorithms A and B with the same growth rate
- hypothesize that running times are ~ c_A f(N) and ~ c_B f(N)
- run algorithms for some value of N, measure running times
- Prediction: Algorithm A is a factor of c_A/c_B faster than Algorithm B

To compare algorithms with different growth rates
- hypothesize that the one with the smaller rate is faster
- validate hypothesis for inputs of interest
  [values of constants may be significant]

To determine whether growth rates are the same or different
- compute ratios of running times as input size doubles
- [growth rates are the same if ratios do not change]

Use algorithms themselves to compute complex leading-term constants

Comparing algorithms with ratio hypothesis

CAUTION
THIS CODE MAY NOT BE READY FOR THE REAL WORLD

```
public class SortCompare
{
   public static void main(String[] args)
   {
      String alg1 = args[0];
      String alg2 = args[1];
      int N  = Integer.parseInt(args[2]);
      int trials = 100;
      if (args.length > 3) trials = Integer.parseInt(args[3]);
      double time1 = 0.0;
      double time2 = 0.0;
      Double[] a1 = new Double[N];
      Double[] a2 = new Double[N];
      for (int t = 0; t < trials; t++)
      {
         for (int i = 0; i < N; i++)
         {  a1[i] = Math.random(); a2[i] = a1[i]; }
         time1 += time(alg1, a1);
         time2 += time(alg2, a2);
      }
      StdOut.printf("For %d random Double values\n    %s is", N, alg1);
      StdOut.printf(" %.1f times faster than %s\n", time2/time1, alg2);

   }
}
```

best to test algs on same input

```
% java SortCompare Insertion Selection 4000
For 4000 random Double values
    Insertion is 1.7 times faster than Selection
```


Summary: turning the crank

Yes, analysis of algorithms might be challenging, BUT

Mathematics might be difficult?
- only a few functions seem to turn up
- doubling, ratio tests cancel complicated constants

Leading term might not be good enough?
- debugging tools are available to identify bottlenecks
- typical programs have short inner loops

Actual data might not match model?
- need to understand input to effectively process it
- approach 1: design for the worst case
- approach 2: randomize, depend on probabilistic guarantee

Timing may be flawed?
- limits on experiments insignificant compared to other sciences
- different computers are different!