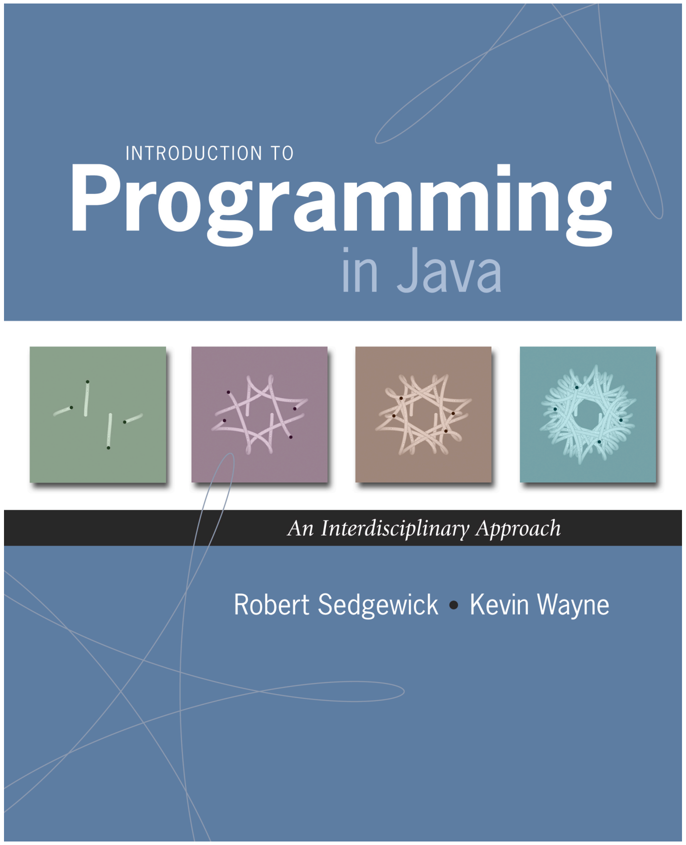
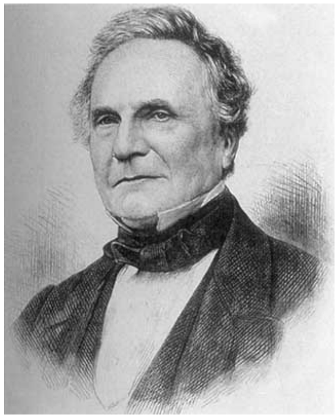
4.1 Performance Analysis

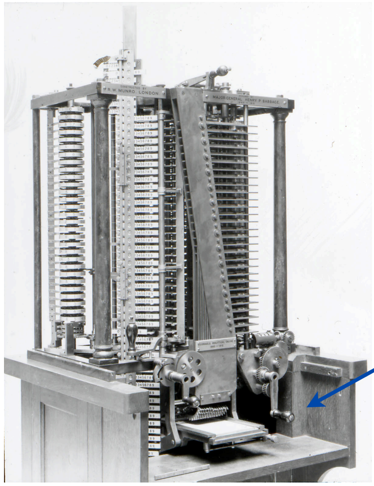


Running Time

"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?

The Challenge

compile

debug
on test cases

solve problems
in practice

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

Key insight. [Knuth 1970s]

Use the scientific method to understand performance.

Scientific Method

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.

Reasons to Analyze Algorithms

Predict performance.

- Will my program finish?
- When will my program finish?

Compare algorithms.

- Will this change make my program faster?
- How can I make my program faster?

Basis for inventing new ways to solve problems.

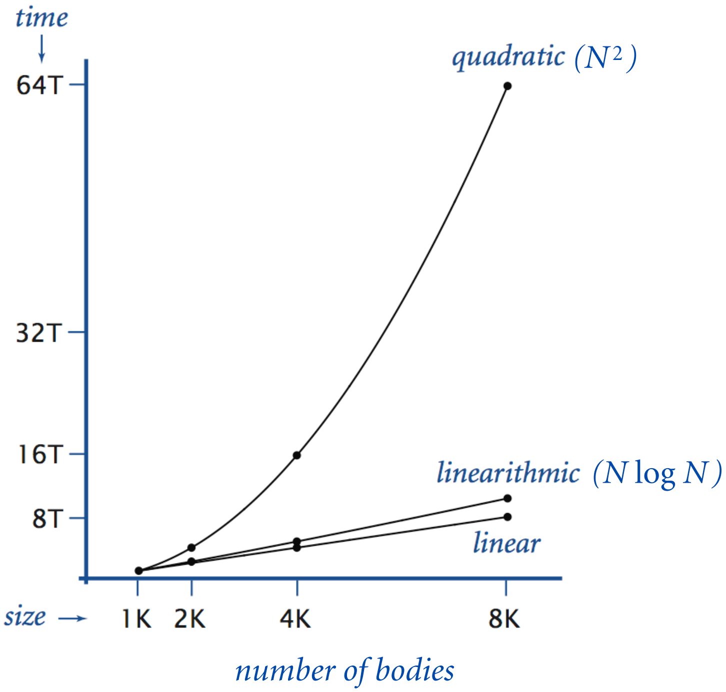
- Enables new technology.
- Enables new research.

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

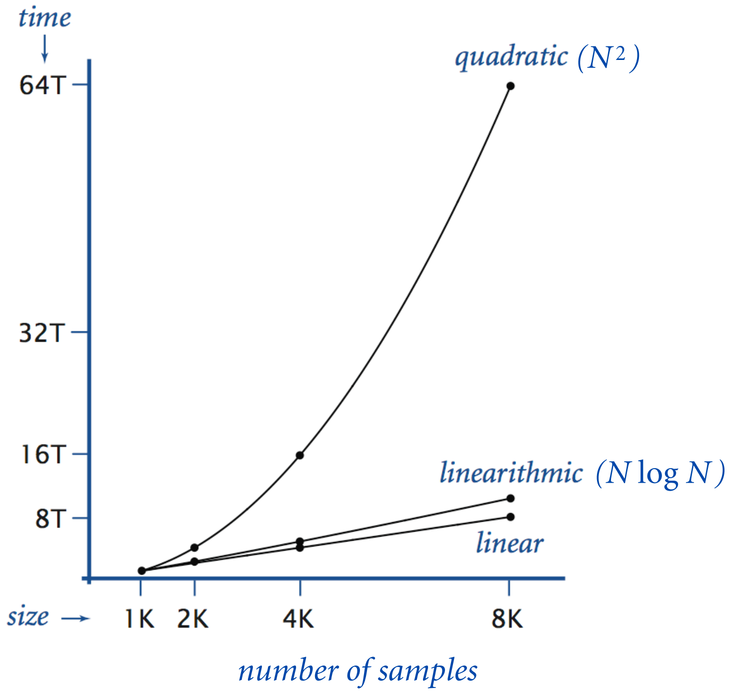


Algorithmic Successes

John Tukey
1965

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.



Example: Three-Sum Problem

Three-sum problem. Given N integers, find triples that sum to 0.

Application. Deeply related to problems in computational geometry.

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

% java ThreeSum < 8ints.txt
 4
 30 -30   0
 30 -20 -10
-30 -10  40
-10   0  10
```

TEQ. Write a program to solve this problem.

Three-Sum

```
public class ThreeSum
{
   // Return number of distinct triples (i, j, k)
   //     such that (a[i] + a[j] + a[k] == 0)
   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)
   {
      int[] a = StdArrayIO.readInt1D();
      int result = count(a);
      StdOut.println(result);
   }
}
```

all possible triples i < j < k

Empirical Analysis

Empirical Analysis

Empirical analysis. Run the program for various input sizes.

N	time (1970) [1]	time (2010) [2]
500	62	0.03
1,000	531	0.26
2,000	4322	2.16
4,000	34377	17.18
8,000	265438	137.76

1. Time in seconds on Jan 18, 2010 running Linux on Sun-Fire-X4100 with 16GB RAM
2. Time in seconds in 1970 running MVT on IBM 360/50 with 256 KB RAM (estimate)

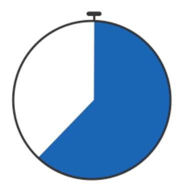
Stopwatch

Q. How to time a program?

A. A stopwatch.

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


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




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

Stopwatch

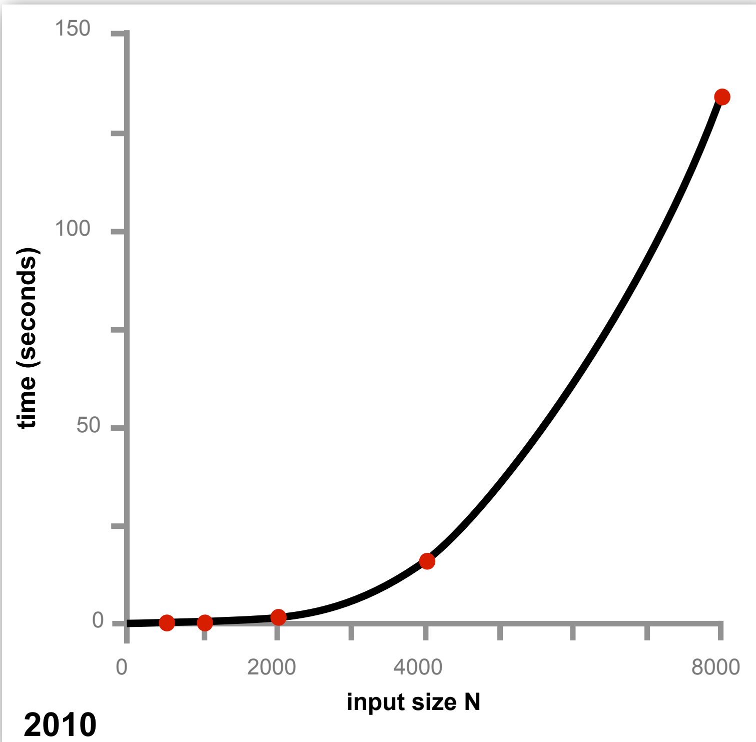
Q. How to time a program?

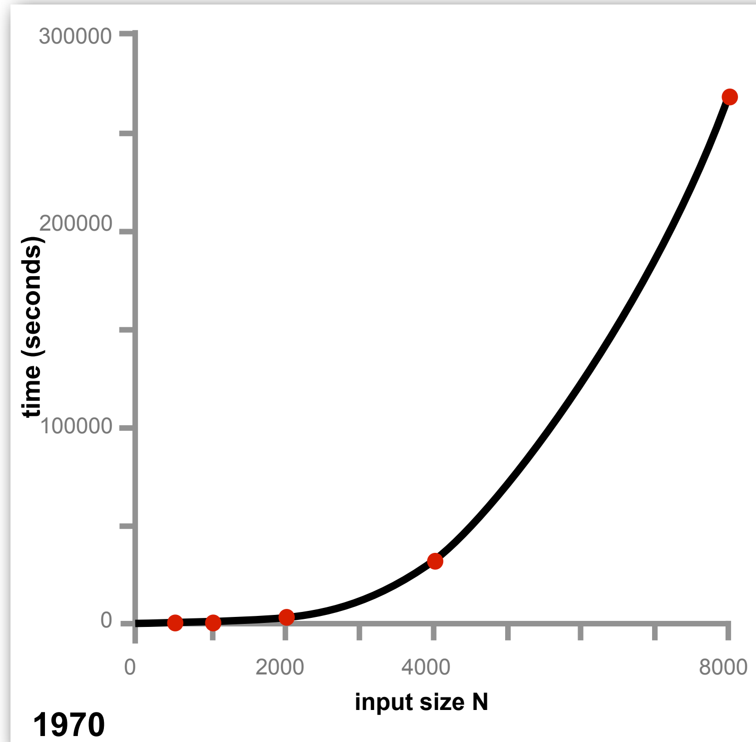
A. Use Java's `System.currentTimeMillis()` method.

```
public static void main(String[] args)
{
    int[] a = StdArrayIO.readInt1D();
    int then = System.currentTimeMillis();
    int result = count(a);
    int now = System.currentTimeMillis();
    StdOut.println(result);
    StdOut.println((now - then)/1000.0);
}
```

Data Analysis

Data analysis. Plot running time vs. input size N.



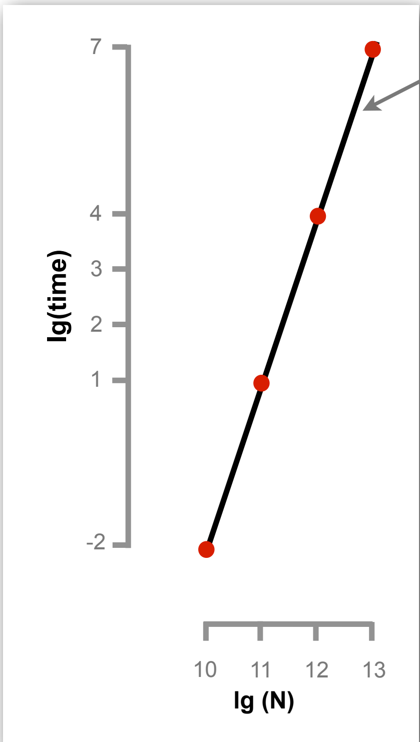


Hypothesis. Running times on different computers differ by a constant factor.

Q. How does running time grow as a function of input size N ?

Data Analysis

Data analysis. Plot running time vs. input size N on a log-log scale



N	T(N)	lg(N)	lg(T(N))
1,000	0.26	10	-1.9
2,000	2.16	11	1.1
4,000	17.18	12	4.1
8,000	137.76	13	7.1

$$\lg(T(N)) = 3 \lg N + a$$

$$T(N) = a N^3$$

Hypothesis: Running time grows as the cube of the input size: $a N^3$

machine-dependent constant factor

Prediction and verification

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

Implicit hypothesis: Running times on different computers differ by a constant factor

Q. How to estimate a ?

A. Solve for it!

$$137.76 = a \times 8000^3$$
$$\Rightarrow \quad a = 2.7 \times 10^{-10}$$

N	T(N)
1,000	0.26
2,000	2.16
4,000	17.18
8,000	137.76

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

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

Observation.

N	time (seconds)
16000	1110.73

validates hypothesis!

Doubling hypothesis

Doubling hypothesis. Quick two-step method for prediction.

Hypothesis: T(2N)/T(N) approaches a constant.

Step 1: Run program, doubling input size, to find the constant

Step 2: Extrapolate to predict next entries

N	T(N)	ratio
500	0.03	-
1,000	0.26	7.88
2,000	2.16	8.43
4,000	17.18	7.96
8,000	137.76	7.96
16,000	1102	8
32,000	8816	8
...	...	
512,000	36112957	

seems to converge to 8

137.76*8

1102*8

$8816*8^4$

Consistent with power law hypothesis

$a(2N)^b / aN^b = 2^b$

(exponent is lg of ratio)

Admits more functions

Ex. $T(N) = N \lg N$

$a(2N \lg 2N) / aN \lg N = 2 + 1/(\lg N) \to 2$

TEQ on Performance 1

Let F(N) be the running time of program **`Mystery`** for input N.

```
public static Mystery
{
    ...
    int N = Integer.parseInt(args[0]);
    ...
}
```

Observation:

N	T(N)	ratio
1,000	4	
2,000	15	4
4,000	60	4
8,000	240	4

Q. Predict the running time for N = 128,000

TEQ on Performance 2

Let F(N) be the running time of program **`Mystery`** for input N.

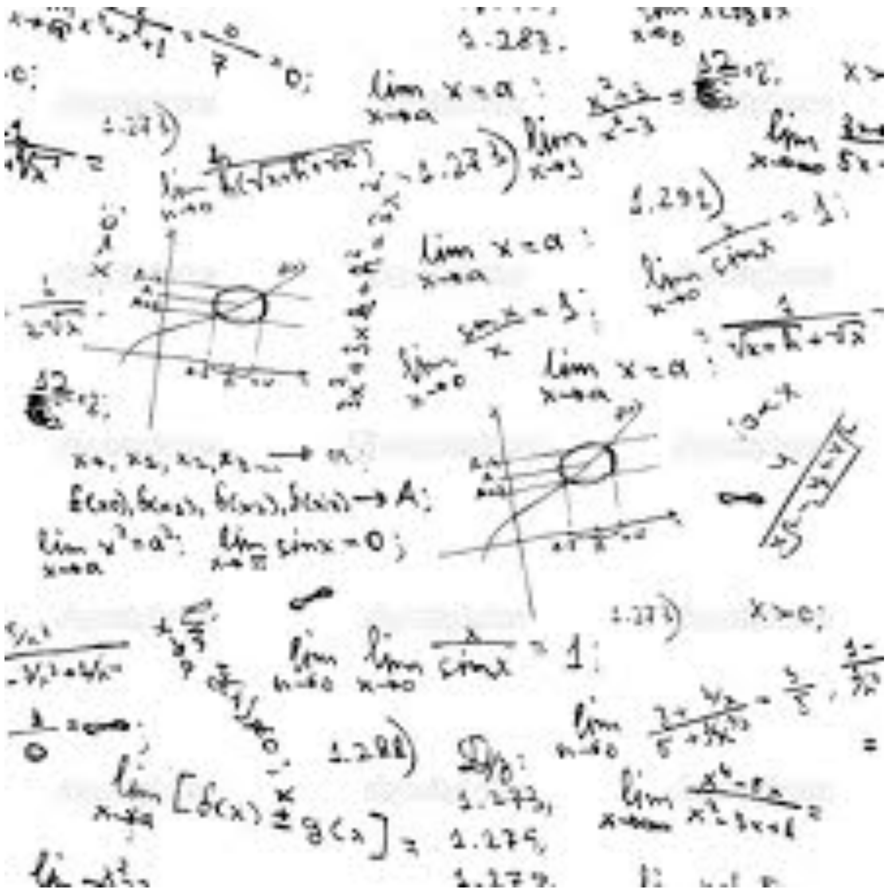
```
public static Mystery
{
   ...
   int N = Integer.parseInt(args[0]);
   ...
}
```

Observation:

N	T(N)	ratio
1,000	4	
2,000	15	4
4,000	60	4
8,000	240	4

Q. Order of growth of the running time?

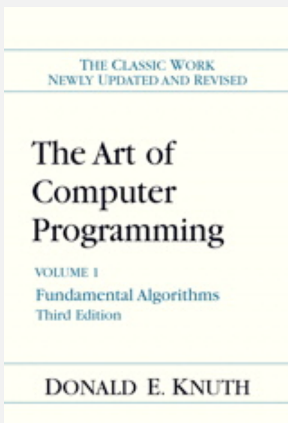
Mathematical Analysis

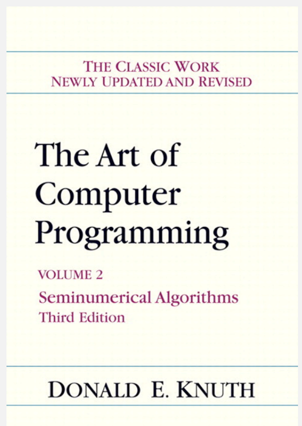


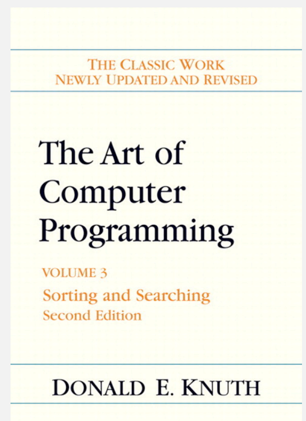
Mathematical models for running time

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.

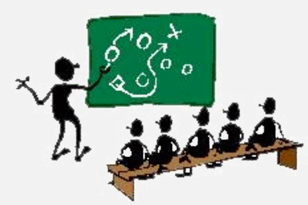






Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.



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 compare	$N + 1$
equal to compare	N
array access	N
increment	$\leq 2N$

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++;
```

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

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

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. $6N^3 + \underbrace{17N^2 \lg N + 7N}_{\text{discard lower-order terms}} \sim 6N^3$

discard lower-order terms
(e.g., N = 1000: 6 billion 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			$\sim c N^2$

depends on input data

depends on machine

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

$\sim N$

$\sim N^2 / 2$

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

may be in inner loop, depends on input data

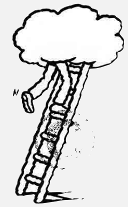
Remark. Focus on instructions in inner loop; ignore everything else!

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

Constants in Power Law

Power law. Running time of a typical program is $\sim a N^b$.

not quite, there may be lg(N) or similar factors

Exponent b depends on: algorithm.

Constant a depends on:

- algorithm
- input data
- hardware (CPU, memory, cache, ...)
- software (compiler, interpreter, garbage collector,...)
- system (network, other applications,...

system independent effects

system dependent effects

Our approach.

- Empirical analysis (doubling hypothesis to determine b, solve for a)
- Mathematical analysis (approximate models based on frequency counts)
- Scientific method (validate models through extrapolation)

Analysis: Empirical vs. Mathematical

Empirical analysis.

- Use doubling hypothesis to solve for a and b in power-law model $\sim a N^b$.
- Easy to perform experiments.
- Model useful for predicting, but not for explaining.

Mathematical analysis.

- Analyze algorithm to develop a model of running time as a function of N [gives a power-law or similar model where doubling hypothesis is valid].
- May require advanced mathematics.
- Model useful for predicting and explaining.

not quite, need empirical study to find a nowadays

Scientific method.

- Mathematical model is independent of a particular machine or compiler; can apply to machines not yet built.
- Empirical analysis is necessary to validate mathematical models.

Order of Growth Classifications

Observation. A small subset of mathematical functions suffice to describe running time of many fundamental algorithms.

```
while (N > 1) {
   N = N / 2;
   ...
}
```

$\lg N$

$\lg N = \log_2 N$

```
for (int i = 0; i < N; i++)
   ...
```

N

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      ...
```

N^2

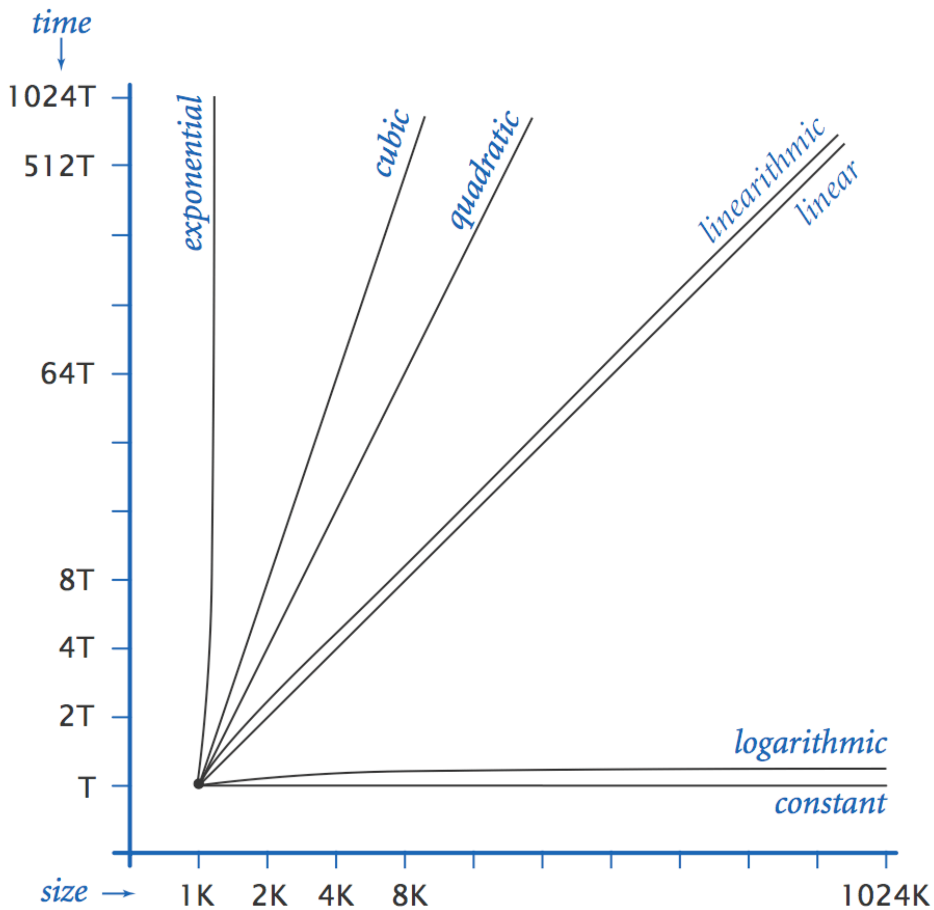
```
public static void g(int N) {
   if (N == 0) return;
   g(N/2);
   g(N/2);
   for (int i = 0; i < N; i++)
      ...
}
```

$N \lg N$

```
public static void f(int N) {
   if (N == 0) return;
   f(N-1);
   f(N-1);
   ...
}
```

2^N

Order of Growth Classifications



Orders of growth (log-log plot)

order of growth		factor for doubling hypothesis
description	function	
constant	1	1
logarithmic	$\log N$	1
linear	N	2
linearithmic	$N \log N$	2
quadratic	N^2	4
cubic	N^3	8
exponential	2^N	2^N

Commonly encountered growth functions

Order of Growth: Consequences

order of growth	*predicted running time if problem size is increased by a factor of 100*
linear	a few minutes
linearithmic	a few minutes
quadratic	several hours
cubic	a few weeks
exponential	forever

Effect of increasing problem size for a program that runs for a few seconds

order of growth	*predicted factor of problem size increase if computer speed is increased by a factor of 10*
linear	10
linearithmic	10
quadratic	3-4
cubic	2-3
exponential	no change

Effect of increasing computer speed on problem size that can be solved in a fixed amount of time

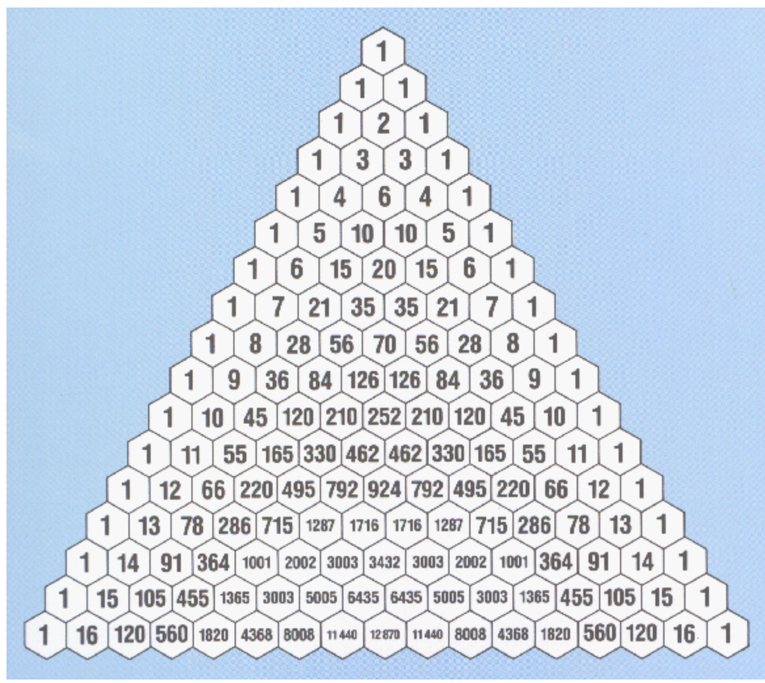
Dynamic Programming

Binomial Coefficients

Binomial coefficient. $\binom{n}{k}$ = number of ways to choose k of n elements.

Pascal's identity.

$$\binom{n}{k} = \underbrace{\binom{n-1}{k-1}}_{\substack{\text{contains} \\ \text{first element}}} + \underbrace{\binom{n-1}{k}}_{\substack{\text{excludes} \\ \text{first element}}}$$

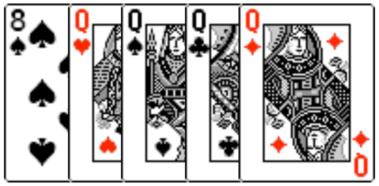


Binomial Coefficients: Poker Odds

Binomial coefficient. $\binom{n}{k}$ = number of ways to choose k of n elements.

Probability of "quads" in Texas hold 'em:

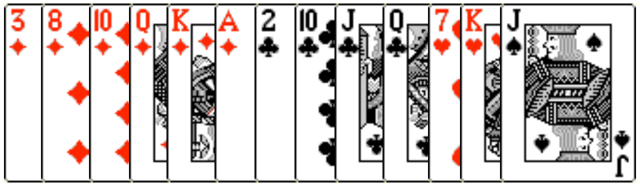
$$\frac{\binom{13}{1} \times \binom{48}{3}}{\binom{52}{7}} = \frac{224,848}{133,784,560} \quad (about \;\; 594 : 1)$$



Probability of 6-4-2-1 split in bridge:

$$\frac{\binom{4}{1} \times \binom{13}{6} \times \binom{3}{1} \times \binom{13}{4} \times \binom{2}{1} \times \binom{13}{2} \times \binom{1}{1} \times \binom{13}{1}}{\binom{52}{13}}$$

$$= \frac{29,858,811,840}{635,013,559,600} \quad (about \;\; 21 : 1)$$



Binomial Coefficients: First Attempt

```
public class SlowBinomial
{
   // Natural recursive implementation
   public static long binomial(long n, long k)
   {
      if (k == 0) return 1;
      if (n == 0) return 0;
      return binomial(n-1, k-1) + binomial(n-1, k);
   }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int K = Integer.parseInt(args[1]);
      StdOut.println(binomial(N, K));
   }

}
```

TEQ on Performance 3

Is this an efficient way to compute binomial coefficients?

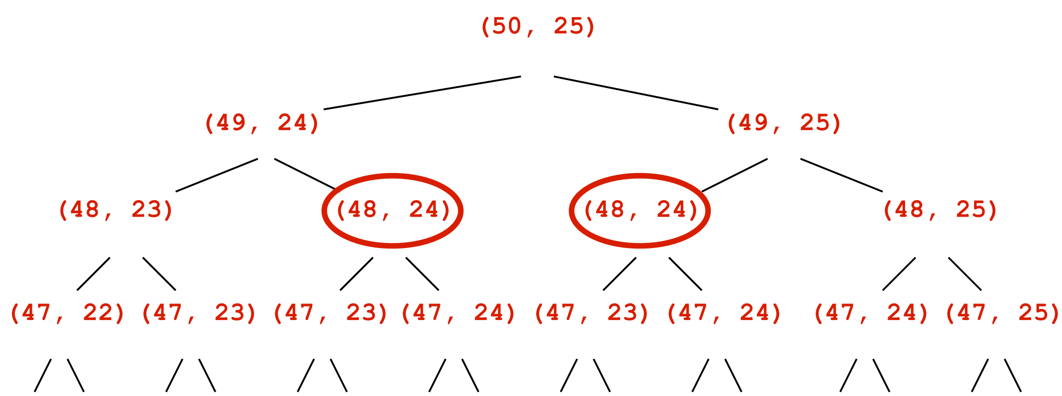
```
public static long binomial(long n, long k)
{
   if (k == 0) return 1;
   if (n == 0) return 0;
   return binomial(n-1, k-1) + binomial(n-1, k);
}
```

TEQ on Performance 3

Is this an efficient way to compute binomial coefficients?

```
public static long binomial(long n, long k)
{
   if (k == 0) return 1;
   if (n == 0) return 0;
   return binomial(n-1, k-1) + binomial(n-1, k);
}
```

A. NO, NO, NO: same essential recomputation flaw as naive Fibonacci.



recursion tree for naïve binomial function

TEQ on Performance 4

Let F(N) be the time to compute binomial(2N, N) using the naive algorithm.

```
public static long binomial(long n, long k)
{
   if (k == 0) return 1;
   if (n == 0) return 0;
   return binomial(n-1, k-1) + binomial(n-1, k);
}
```

Observation: F(N+1)/F(N) is about 4.

What is the order of growth of the running time?

Dynamic Programming

Key idea. Save solutions to subproblems to avoid recomputation.

k

n \ *k*	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15

binomial(n, k)

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$20 = 10 + 10$

Tradeoff. Trade (a little) memory for (a huge amount of) time.

Binomial Coefficients: Dynamic Programming

```
public class Binomial
{
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      int K = Integer.parseInt(args[1]);
      long[][] bin = new long[N+1][K+1];

      // base cases
      for (int k = 1; k <= K; k++) bin[0][K] = 0;
      for (int n = 0; n <= N; n++) bin[N][0] = 1;

      // bottom-up dynamic programming
      for (int n = 1; n <= N; n++)
         for (int k = 1; k <= K; k++)
            bin[n][k] = bin[n-1][k-1] + bin[n-1][k];

      // print results
      StdOut.println(bin[N][K]);
   }
}
```

TEQ on Performance 5

Let F(N) be the time to compute binomial(2N, N) using dynamic programming.

```
for (int n = 1; n <= 2*N; n++)
   for (int k = 1; k <= N; k++)
      bin[n][k] = bin[n-1][k-1] + bin[n-1][k];
```

What is the order of growth of the running time?

In the real world: Stirling's Approximation

Why not use the formula to compute binomial coeffiecients? $\binom{n}{k} = \frac{n!}{n!\,(n-k)!}$

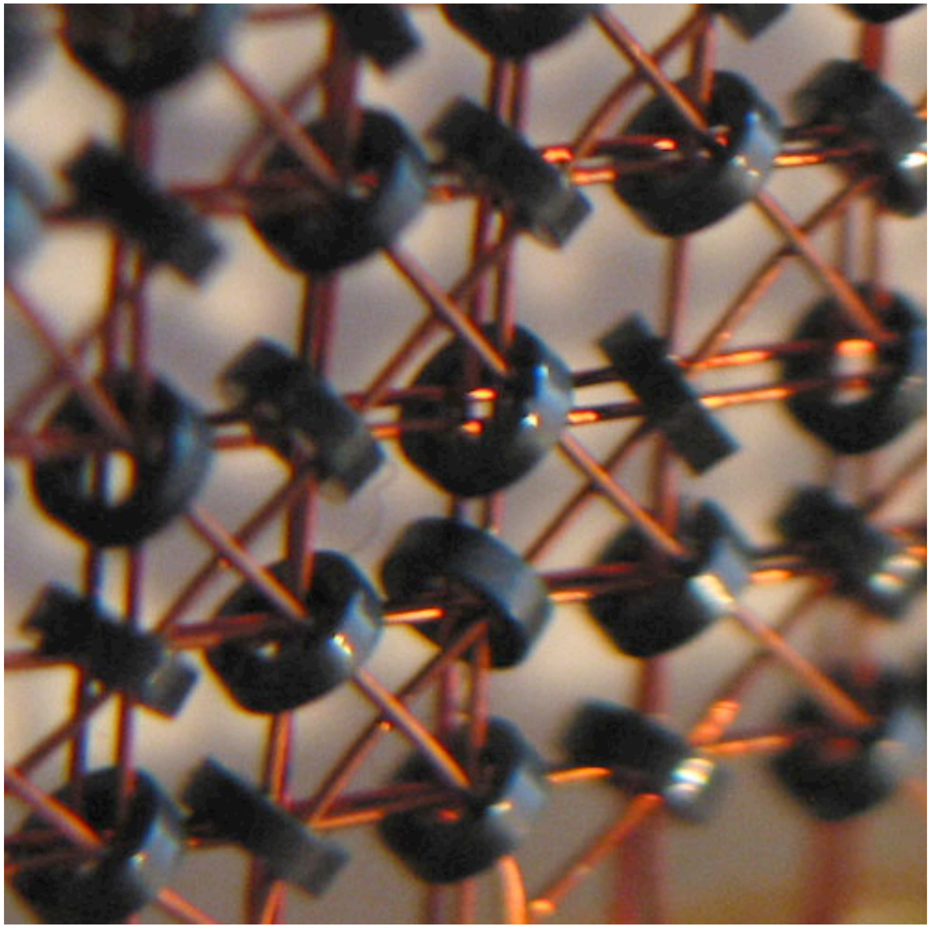
Doesn't work: 52! overflows a `long`, even though final result doesn't.

Instead of computing exact values, use Stirling's approximation:

$$\ln n! \approx n \ln n - n + \frac{\ln(2\pi n)}{2} + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5}$$

approach	order of growth of running time	comment
recursive	2^N	useless unless N is very small
dynamic programming	N^2	best way to get exact answer
direct from formula	N	no good for large N (overflow)
Stirling's approximation	constant	extremely accurate in practice

Memory



Typical Memory Requirements for Java Data Types

Bit. 0 or 1.

Byte. 8 bits.

Megabyte (MB). 2^{10} bytes ~ 1 million bytes.

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

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

type	*bytes*
`int[]`	$4N + 16$
`double[]`	$8N + 16$
`Charge[]`	$36N + 16$
`int[][]`	$4N^2 + 20N + 16$
`double[][]`	$8N^2 + 20N + 16$
`String`	$2N + 40$

typical computer '10 has about 2GB memory

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

TEQ on Performance 6

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]++;
      }
   }
}
```

Summary

Q. How can I evaluate the performance of my program?

A. Computational experiments, mathematical analysis, scientific method

Q. What if it's not fast enough? Not enough memory?

- Understand why.
- Buy a faster computer.
- Learn a better algorithm (COS 226, COS 423).
- Discover a new algorithm.

attribute	better machine	better algorithm
cost	$$$ or more.	$ or less.
applicability	makes "everything" run faster	does not apply to some problems
improvement	incremental quantitative improvements expected	dramatic qualitative improvements possible