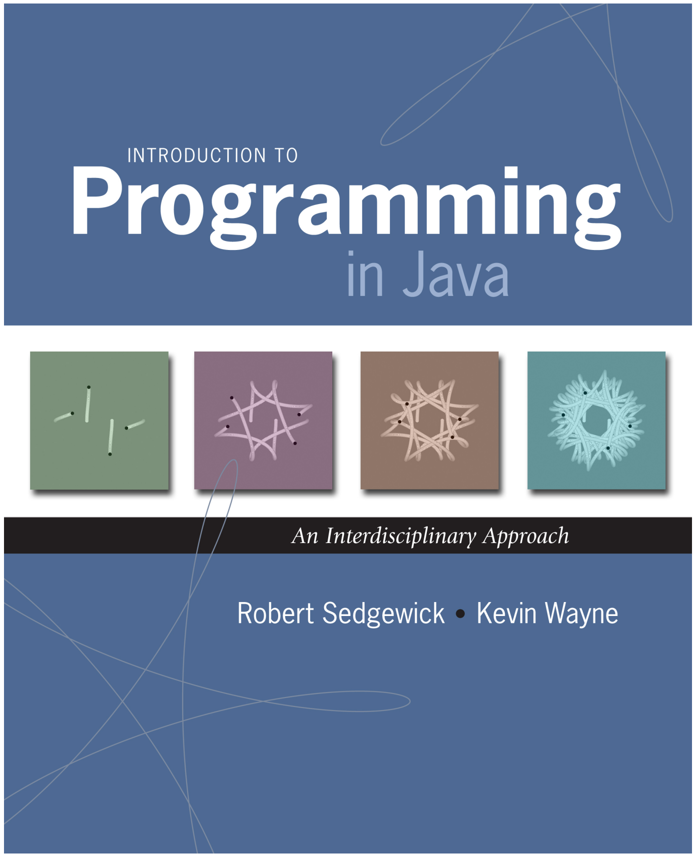
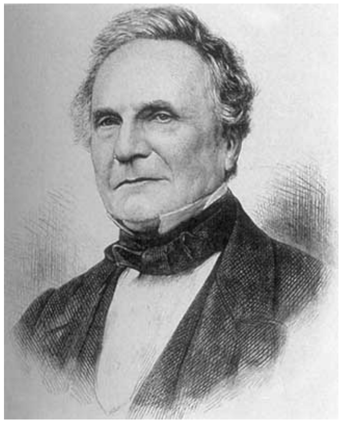
4.1 Performance



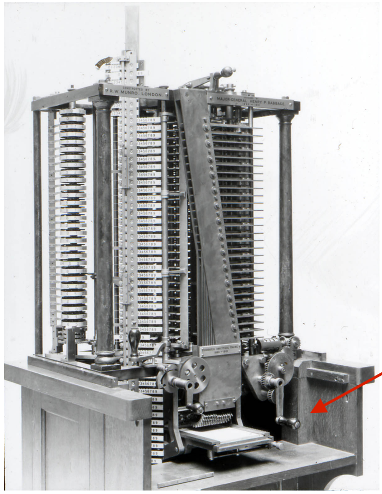
Copyright © 2002–2010

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

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

compile debug solve problems in practice

Key insight. [Knuth 1970s]
Use the scientific method to understand performance.

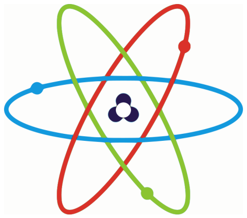
Scientific Method

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.
- Hypothesis 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?

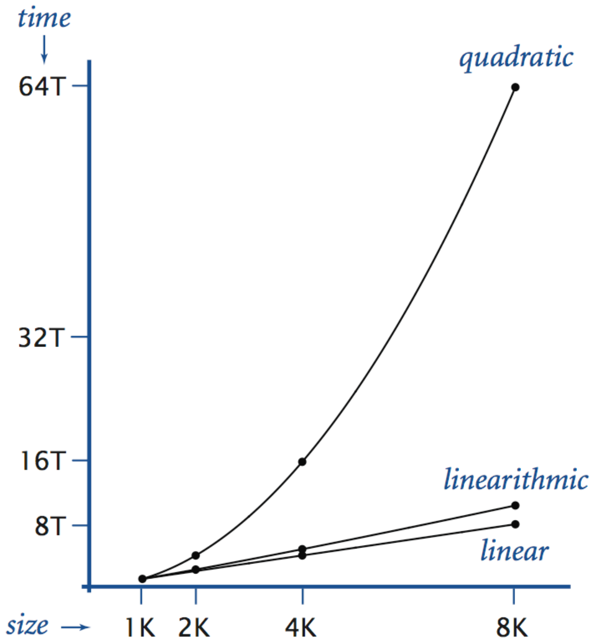
Basis for inventing new ways to solve problems.
- Enables new technology.
- Enables new research.

Algorithmic Successes

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

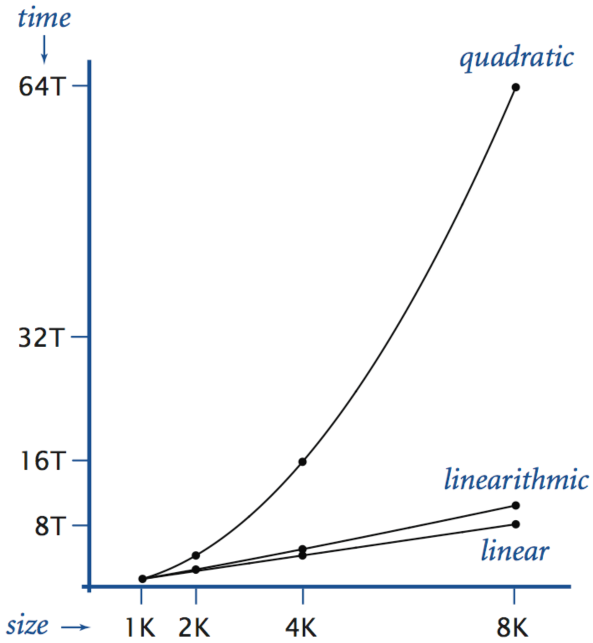


Algorithmic Successes

N-body Simulation.

- Simulate gravitational interactions among N bodies.
- Application: cosmology, semiconductors, fluid dynamics, …
- Brute force: N^2 steps.
- Barnes-Hut algorithm: $N \log N$ steps, enables new research.

Andrew Appel
PU '81



Three-Sum Problem

Three-sum problem. Given N integers, how many triples sum to 0 ?

Context. 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
```

Q. How would you write a program to solve the problem?

Three-Sum: Brute-Force Solution

```
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) {
      int[] a = StdArrayIO.readInt1D();
      StdOut.println(count(a));
   }
}
```

all possible triples i < j < k such that a[i] + a[j] + a[k] = 0

Empirical Analysis

Empirical Analysis

Empirical analysis. Run the program for various input sizes.

N	*time* †
512	0.03
1024	0.26
2048	2.16
4096	17.18
8192	136.76

† Running Linux on Sun-Fire-X4100 with 16GB RAM

Caveat. If N is too small, you will measure mainly noise.

Stopwatch

Q. How to time a program?

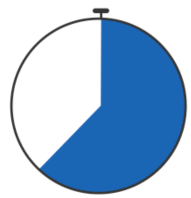
A. A stopwatch.


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

tick tick tick

```
0
```

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



tick tick tick tick tick tick
tick tick tick tick tick tick
tick tick tick tick tick tick
tick tick tick tick tick tick

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

Stopwatch

Q. How to time a program?

A. A `Stopwatch` object.

public class Stopwatch

	`Stopwatch()`	*create a new stopwatch and start it running*
`double`	`elapsedTime()`	*return the elapsed time since creation, in seconds*

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

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

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

Stopwatch

Q. How to time a program?

A. A **Stopwatch** object.

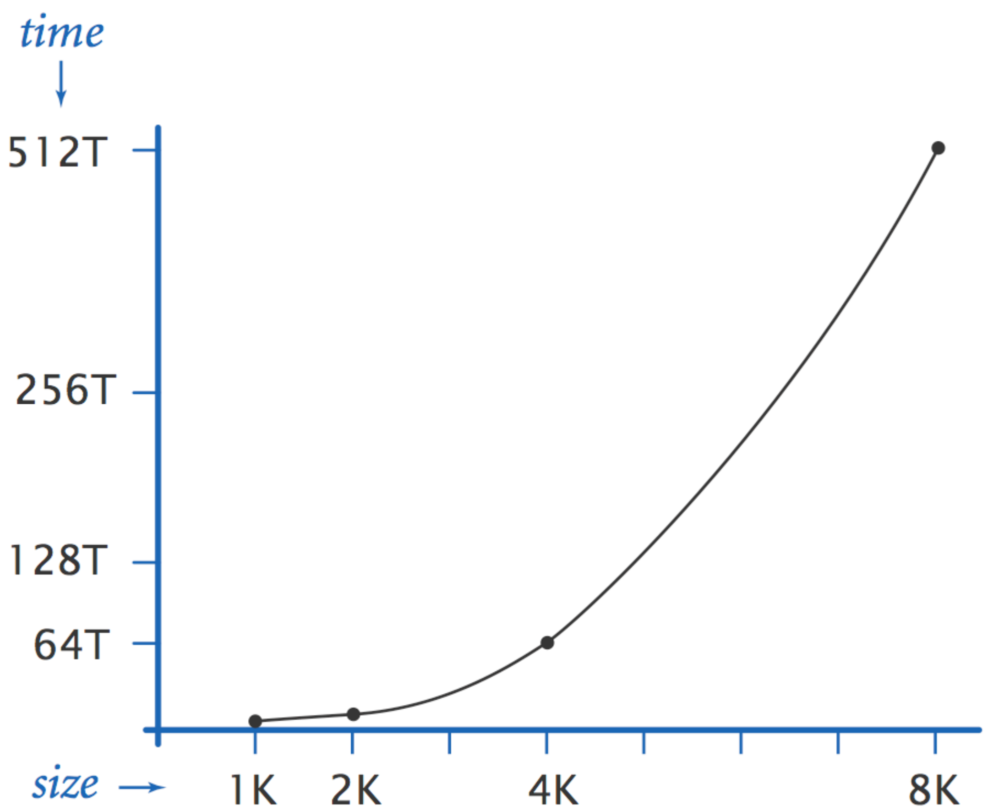
public class Stopwatch

	`Stopwatch()`	*create a new stopwatch and start it running*
`double`	`elapsedTime()`	*return the elapsed time since creation, in seconds*

```
public static void main(String[] args) {
   int[] a = StdArrayIO.readInt1D();
   Stopwatch timer = new Stopwatch();
   StdOut.println(count(a));
   StdOut.println(timer.elapsedTime());
}
```

Empirical Analysis

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



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

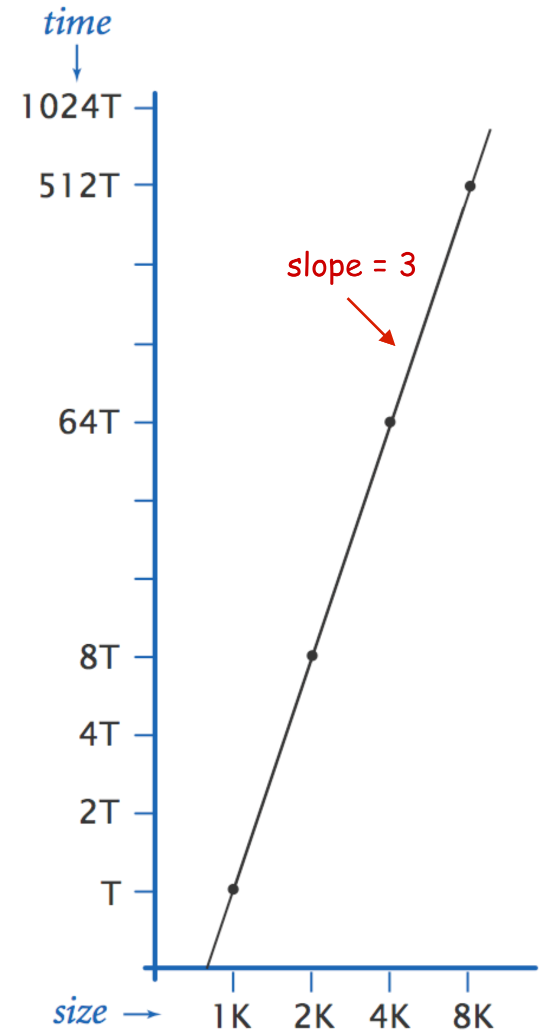
Empirical Analysis

Initial hypothesis. Running time approximately obeys a power law $T(N) = a\,N^{b}$.

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

Consequence. Power law yields straight line.

slope = b



Refined hypothesis. Running time grows as cube of input size: $a\,N^{3}$.

slope

Doubling Hypothesis

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

Run program, doubling the size of the input?

N	*time* †	*ratio*
512	0.033	-
1024	0.26	7.88
2048	2.16	8.43
4096	17.18	7.96
8192	136.76	7.96

seems to converge to a constant $c = 8$

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

Performance Challenge 1

Let $T(N)$ be running time of **`main()`** as a function of input size N.

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

Scenario 1. $T(2N) / T(N)$ converges to about 4.

Q. What is order of growth of the running time?

$1 \quad N \quad N^2 \quad N^3 \quad N^4 \quad 2^N$

Performance Challenge 2

Let $T(N)$ be running time of **`main()`** as a function of input N.

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

Scenario 2. $T(2N) / T(N)$ converges to about 2.

Q. What is order of growth of the running time?

$1 \quad N \quad N^2 \quad N^3 \quad N^4 \quad 2^N$

Prediction and Validation

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

Q. How to estimate a?
A. Run the program!

N	*time* †
4096	17.18
4096	17.15
4096	17.17

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

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* †
16384	1118.86

← validates hypothesis

Mathematical Analysis

Donald Knuth
Turing award '74

Mathematical Analysis

Running time. Count up frequency of execution of each instruction and weight by its execution time.

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

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

between N (no zeros)
and 2N (all zeros)

Mathematical Analysis

Running time. Count up frequency of execution of each instruction and weight by its execution time.

```
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	$N+2$
variable assignment	$N+2$
less than comparison	$1/2\ (N+1)\ (N+2)$
equal to comparison	$1/2\ N\ (N-1)$
array access	$N\ (N-1)$
increment	$\leq\ N^2$

$$0 + 1 + 2 + \ldots + (N-1) = 1/2\ \ N(N-1)$$

becoming very tedious to count

Tilde Notation

Tilde notation.

- Estimate running time 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. $6N^3 + 17N^2 + 56 \sim 6N^3$

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

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

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

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

Mathematical Analysis

Running time. Count up frequency of execution of each instruction and weight by its execution time.

```
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;
}
```

1

N

$\sim N^2/2$

$\sim N^3/6$

inner loop

depends on input data

Inner loop. Focus on instructions in "inner loop."

Constants in Power Law

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

Exponent b depends on: algorithm.

Leading constant a depends on:

- Algorithm.
- Input data.
- Caching.
- Machine.
- Compiler.
- Garbage collection.
- Just-in-time compilation.
- CPU use by other applications.

system independent effects

system dependent effects

Our approach. Use doubling hypothesis (or mathematical analysis) to estimate exponent b, run experiments to estimate a.

Analysis: Empirical vs. Mathematical

Empirical analysis.

- Measure running times, plot, and fit curve.
- Easy to perform experiments.
- Model useful for predicting, but not for explaining.

Mathematical analysis.

- Analyze algorithm to estimate # ops as a function of input size.
- May require advanced mathematics.
- Model useful for predicting and explaining.

Critical difference. Mathematical analysis is independent of a particular machine or compiler; applies to machines not yet built.

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

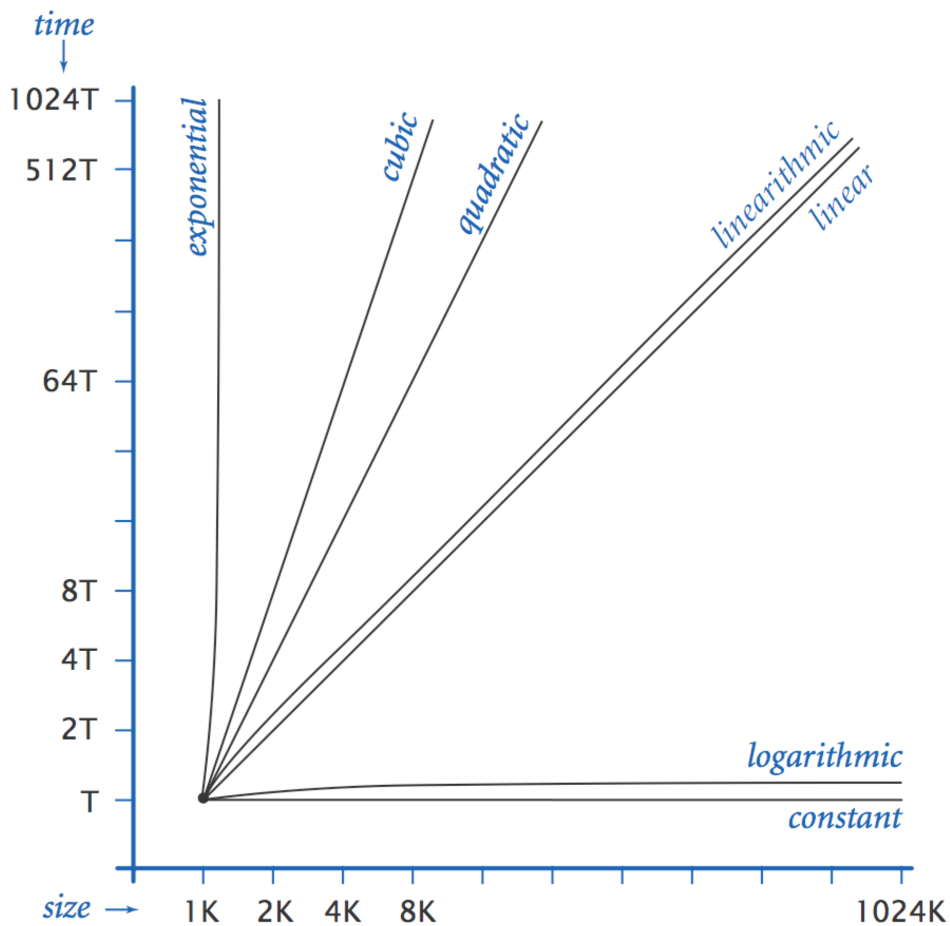
```
public static void g(int N) {
   if (N == 0) return;
   g(N/2);
   g(N/2);
   for (int i = 0; i < N; i++)
      ...
}
```

$N \lg N$

```
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 description	*order of growth* function	*factor for doubling hypothesis*
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	1

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

Binary Search

Sequential Search vs. Binary Search

Sequential search in an unordered array.

- Examine each entry until finding a match (or reaching the end).
- Takes time proportional to length of array in worst case.

43	72	13	84	64	33	97	51	6	25	95	96	53	14	93

Binary search in an ordered array.

- Examine the middle entry.
- If equal, return index.
- If too large, search in left half (recursively).
- If too small, search in right half (recursively).

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search: Java Implementation

Invariant. If **`key`** appears in the array, then **`a[lo]`** ≤ **`key`** ≤ **`a[hi]`**.

```
// precondition: array a[] is sorted
public static int search(int key, int[] a) {
   int lo = 0;
   int hi = a.length - 1;
   while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else return mid;
   }
   return -1;  // not found
}
```

Java library implementation. **`Arrays.binarySearch()`**.

Binary Search: Mathematical Analysis

Proposition. Binary search in an ordered array of size N takes at most $1 + \log_2 N$ 3-way compares.

Pf. After each 3-way compare, problem size decreases by a factor of 2.

$$N \to N/2 \to N/4 \to N/8 \to \ldots \to 1$$

Q. How many times can you divide N by 2 until you reach 1?

A. About $\log_2 N$.

$$\begin{array}{c}
1 \\
2 \to 1 \\
4 \to 2 \to 1 \\
8 \to 4 \to 2 \to 1 \\
16 \to 8 \to 4 \to 2 \to 1 \\
32 \to 16 \to 8 \to 4 \to 2 \to 1 \\
64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 \\
128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 \\
256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 \\
512 \to 256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1 \\
1024 \to 512 \to 256 \to 128 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1
\end{array}$$

Memory


elixir
M2U51264DS8HC3G-5T
512MB DDR-400MHz-CL3
PC3200U-30331
Warranty Void If Removed
0542.MN05A1107A.5YM0410N08.XX.CN

Typical Memory Requirements for Primitive Types

Bit. 0 or 1.
Byte. 8 bits.
Megabyte (MB). 1 million bytes ~ 2^{10} bytes.
Gigabyte (GB). 1 billion bytes ~ 2^{20} bytes.

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

Q. How much memory (in bytes) does your computer have?

Typical Memory Requirements for Reference Types

Memory of an object.

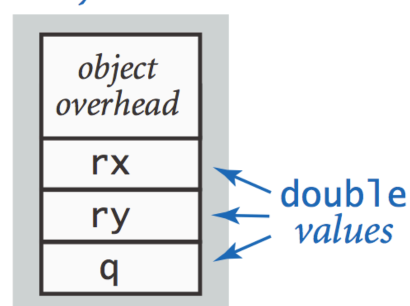
- Memory for each instance variable, plus
- Object overhead = 8 bytes on a 32-bit machine.

16 bytes on a 64-bit machine

Charge object

```
public class Charge
{
   private double rx;
   private double ry;
   private double q;
...
}
```

32 bytes



Memory of a reference. 4 byte pointer on a 32-bit machine.

8 bytes on a 64-bit machine

Typical Memory Requirements for Array Types

Memory of an array.

- Memory for each array entry.
- Array overhead = 16 bytes on a 32-bit machine.

24 bytes on a 64-bit machine

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

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

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 or more memory.
- Learn a better algorithm. ← see COS 226
- Discover a new algorithm. ← see COS 423

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