Cast of characters

Programmer needs to develop a working solution.

Client wants to solve problem efficiently.

Student might play any or all of these roles someday.

Theoretician wants to understand.

Basic blocking and tackling is sometimes necessary. [this lecture]

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 (1864)
Reasons to analyze algorithms

- Predict performance.
- Compare algorithms. (this course (BBM 202))
- Provide guarantees.
- Understand theoretical basis. (Analysis of algorithms (BBM 408))

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.

- sFFT: Sparse Fast Fourier Transform algorithm (Hassanieh et al., 2012)
  - A faster Fourier Transform: $k \log N$ steps (with $k$ sparse coefficients)

Some algorithmic successes

N-body simulation.
- Simulate gravitational interactions among $N$ bodies.
- Brute force: $N^2$ steps.
- Barnes-Hut algorithm: $N \log N$ steps, enables new research.

The challenge

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

Why is my program so slow? Why does it run out of memory?

Key insight. [Knuth 1970s] Use scientific method to understand performance.
Scientific method applied to analysis of algorithms

A framework for predicting performance and comparing algorithms.

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.

Feature of the natural world = computer itself.

Example: 3-sum

3-sum. Given N distinct integers, how many triples sum to exactly zero?

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>-40</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>-20</td>
<td>-10</td>
</tr>
<tr>
<td>3</td>
<td>-40</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Context. Deeply related to problems in computational geometry.

3-sum: brute-force algorithm

```java
public class ThreeSum {
    public static int count(int[] a) {
        int N = a.length;
        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++;
        return count;
    }
    public static void main(String[] args) {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

3-sum: brute-force algorithm

- Observations
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- Memory
Measuring the running time

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

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

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>7</td>
</tr>
</tbody>
</table>

Measuring the running time

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

```java
public class Stopwatch
{    // implementation (part of stdlib.jar)
    public class Stopwatch()
    {        // create a new stopwatch
        public double elapsedTime()
        {            // time since creation (in seconds)
            return (System.currentTimeMillis() - start) / 1000.0;
        }
    }
}
```

```java
public class Stopwatch
{    // implementation (part of stdlib.jar)

    public class Stopwatch()
    {        // create a new stopwatch
        public double elapsedTime()
        {            // time since creation (in seconds)
            return (System.currentTimeMillis() - start) / 1000.0;
        }
    }
}
```

Empirical analysis

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

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

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>7</td>
</tr>
</tbody>
</table>
Data analysis

**Standard plot.** Plot running time $T(N)$ vs. input size $N$.

![Standard plot](image)

**Log-log plot.** Plot running time $T(N)$ vs. input size $N$ using log-log scale.

![Log-log plot](image)

**Regression.** Fit straight line through data points: $a N^b$ where $a \approx 2.999$.

**Prediction and validation**

**Hypothesis.** The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

**Predictions.**
- 51.0 seconds for $N = 8,000$.
- 408.1 seconds for $N = 16,000$.

**Observations.**

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.000</td>
<td>51.1</td>
</tr>
<tr>
<td>8.000</td>
<td>51</td>
</tr>
<tr>
<td>8.000</td>
<td>51.1</td>
</tr>
<tr>
<td>16.000</td>
<td>410.8</td>
</tr>
</tbody>
</table>

*validates hypothesis!

**Doubling hypothesis**

**Doubling hypothesis.** Quick way to estimate $b$ in a power-law relationship.

Run program, doubling the size of the input.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
<th>ratio</th>
<th>$\log$ ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0</td>
<td>4.8</td>
<td>2.3</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>6.9</td>
<td>2.8</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>7.7</td>
<td>2.9</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

seems to converge to a constant $b = 3$

**Hypothesis.** Running time is about $a N^b$ with $b = \log$ ratio.

**Caveat.** Cannot identify logarithmic factors with doubling hypothesis.
Doubling hypothesis

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

Q. How to estimate $a$ (assuming we know $b$)?
A. Run the program (for a sufficient large value of $N$) and solve for $a$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.000</td>
<td>51.1</td>
</tr>
<tr>
<td>8.000</td>
<td>51</td>
</tr>
<tr>
<td>8.000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

Hypothesis. Running time is about $0.998 \times 10^{-10} \times N^3$ seconds.

Experimental algorithmics

System independent effects.
- Algorithm.
- Input data.
  
System dependent effects.
- Hardware: CPU, memory, cache, …
- Software: compiler, interpreter, garbage collector, …
- System: operating system, network, other applications, …

Bad news. Difficult to get precise measurements.
Good news. Much easier and cheaper than other sciences.

In practice, constant factors matter too!

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

```java
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] = ...;
...
```

<table>
<thead>
<tr>
<th>$N$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>0.33</td>
</tr>
<tr>
<td>2.000</td>
<td>0.35</td>
</tr>
<tr>
<td>4.000</td>
<td>1.6</td>
</tr>
<tr>
<td>8.000</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Jenny ~ $c_1 N^2$ seconds

Kenny ~ $c_2 N^2$ seconds

Analysis of Algorithms

- Observations
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- Memory
Mathematical models for running time

**Total running time**: sum of cost \( \times \) 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>variable declaration</td>
<td>int a</td>
<td>c</td>
</tr>
<tr>
<td>assignment statement</td>
<td>a = b</td>
<td>c</td>
</tr>
<tr>
<td>integer compare</td>
<td>a &lt; b</td>
<td>c</td>
</tr>
<tr>
<td>array element access</td>
<td>a[i]</td>
<td>c</td>
</tr>
<tr>
<td>array length</td>
<td>a.length</td>
<td>c</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[N]</td>
<td>c_1</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[N][N]</td>
<td>c_2</td>
</tr>
<tr>
<td>string length</td>
<td>a.length()</td>
<td>c</td>
</tr>
<tr>
<td>substring extraction</td>
<td>a.substring(N/2, N)</td>
<td>c</td>
</tr>
<tr>
<td>string concatenation</td>
<td>a + t</td>
<td>c_10</td>
</tr>
</tbody>
</table>

Novice mistake. Abusive string concatenation.

Example: 1-sum

Q. How many instructions as a function of input size \( N \)?

```java
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>2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>N + 1</td>
</tr>
<tr>
<td>equal to compare</td>
<td>N</td>
</tr>
<tr>
<td>array access</td>
<td>N</td>
</tr>
<tr>
<td>increment</td>
<td>N to 2 N</td>
</tr>
</tbody>
</table>
Example: 2-sum

Q. How many instructions as a function of input size 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 + ... + (N - 1) = \frac{1}{2} N(N - 1)

Simplifying the calculations

“...if we consider the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings.” — Alan Turing

Technical definition. \( f(N) = g(N) \) means \( \lim_{N \to \infty} \frac{f(N)}{g(N)} = 1 \)
Simplification 2: 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!

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>tilde notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>( N + 2 )</td>
<td>( \sim N )</td>
</tr>
<tr>
<td>assignment statement</td>
<td>( N + 2 )</td>
<td>( \sim N )</td>
</tr>
<tr>
<td>less than compare</td>
<td>( \frac{1}{2} (N + 1)(N + 2) )</td>
<td>( \sim \frac{1}{2} N )</td>
</tr>
<tr>
<td>equal to compare</td>
<td>( \frac{1}{2} N (N - 1) )</td>
<td>( \sim \frac{1}{2} N )</td>
</tr>
<tr>
<td>array access</td>
<td>( N (N - 1) )</td>
<td>( \sim N )</td>
</tr>
<tr>
<td>increment</td>
<td>( \frac{1}{2} N (N - 1) ) to ( N (N - 1) )</td>
<td>( \sim \frac{1}{2} N )</td>
</tr>
</tbody>
</table>

Example: 2-sum

Q. Approximately how many array accesses as a function of input size \( N \)?

A. \( \sim N^2 \) array accesses.

Bottom line. Use cost model and tilde notation to simplify frequency counts.

Example: 3-sum

Q. Approximately how many array accesses as a function of input size \( N \)?

A. \( \sim \frac{1}{2} N^3 \) array accesses.

Bottom line. Use cost model and tilde notation to simplify frequency counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?
A1. Take discrete mathematics course.
A2. Replace the sum with an integral, and use calculus!

Ex 1. \( 1 + 2 + \ldots + N \).
\[
\sum_{i=1}^{N} i \sim \int_{0}^{N} x \, dx = \frac{1}{2} N^2
\]

Ex 2. \( 1 + 1/2 + 1/3 + \ldots + 1/N \).
\[
\sum_{i=1}^{N} \frac{1}{i} \sim \int_{1}^{N} \frac{1}{x} \, dx = \ln N
\]

Ex 3. 3-sum triple loop.
\[
\sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} 1 \sim \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{1}{xyz} \, dz \, dy \, dx \sim \frac{1}{6} N^3
\]
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.

Bottom line. We use approximate models in this course: \( T(N) \sim c N^3 \).

Common order-of-growth classifications

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.

<table>
<thead>
<tr>
<th>order of growth</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
<th>( T(2N) / T(N) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td>while ((N &gt; 1)) { ( N = N / 2 ); ... }</td>
<td>divide in half</td>
<td>binary search</td>
<td>( \sim 1 )</td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td>for ((i = 0; 1 &lt; N; i++)) { ... }</td>
<td>loop</td>
<td>find the maximum</td>
<td>2</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>linearithmic</td>
<td>[see mergesort lecture]</td>
<td>divide and conquer</td>
<td>mergesort</td>
<td>( \sim 2 )</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>quadratic</td>
<td>for ((i = 0; i &lt; N; i++)) for ((j = 0; j &lt; N; j++)) { ... }</td>
<td>double loop</td>
<td>check all pairs</td>
<td>4</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>cubic</td>
<td>for ((i = 0; i &lt; N; i++)) for ((j = 0; j &lt; N; j++)) for ((k = 0; k &lt; N; k++)) { ... }</td>
<td>triple loop</td>
<td>check all triples</td>
<td>8</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>exponential</td>
<td>[see combinatorial search lecture]</td>
<td>exhaustive search</td>
<td>check all subsets</td>
<td>( T(N) )</td>
</tr>
</tbody>
</table>
Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1970s</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
</tr>
<tr>
<td>N</td>
<td>hundreds</td>
</tr>
<tr>
<td>N</td>
<td>hundred</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Practical implications of order-of-growth

<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>
<th>time for 100x more data</th>
<th>size for 100x faster computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>independent of input size</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td>nearly independent of input size</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td>optimal for N inputs</td>
<td>a few minutes</td>
<td>100x</td>
<td>-</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>nearly optimal for N inputs</td>
<td>a few minutes</td>
<td>100x</td>
<td>-</td>
</tr>
<tr>
<td>N</td>
<td>quadratic</td>
<td>not practical for large problems</td>
<td>several hours</td>
<td>10x</td>
<td>-</td>
</tr>
<tr>
<td>N</td>
<td>cubic</td>
<td>not practical for medium problems</td>
<td>several weeks</td>
<td>4-5x</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>exponential</td>
<td>useful only for tiny problems</td>
<td>forever</td>
<td>1x</td>
<td>-</td>
</tr>
</tbody>
</table>

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.
- Too small, go left.
- Too big, go right.
- Equal, found.

6 13 14 25 33 43 51 93 64 72 84 93 95 96 97
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
↓  ↑
lo  hi mid
Goal. Given a sorted array and a key, find index of the key in the array?

Successful search. Binary search for 33.

lo = hi

return 4
**Binary search demo**

*Goal.* Given a sorted array and a key, find index of the key in the array?

*Unsuccessful search.* Binary search for 34.

```
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
```

- `lo` = hi

```
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
```

- `lo` = hi
- return -1
Binary search: Java implementation

Trivial to implement?
• First binary search published in 1946; first bug-free one published in 1962.
• Bug in Java’s Arrays.binarySearch() discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, 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;
}
```

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

Binary search: mathematical analysis

Proposition. Binary search uses at most 1 + \lg N \text{ compares to search in a sorted array of size } N.

Def. \( T(N) \equiv \# \text{ compares to binary search in a sorted subarray of size at most } N. \)

Binary search recurrence. \( T(N) \leq T(N / 2) + 1 \) for \( N > 1 \), with \( T(1) = 1 \).

Pf sketch.

\[
T(N) \leq T(N / 2) + 1 \\
\leq T(N / 4) + 1 + 1 \\
\leq T(N / 8) + 1 + 1 + 1 \\
\ldots \\
\leq T(N / N) + 1 + 1 + \ldots + 1 \\
= 1 + \lg N
\]

An \( N^2 \log N \) algorithm for 3-sum

Algorithm.
• Sort the \( N \) (distinct) numbers.
• For each pair of numbers \( a[i] \) and \( a[j] \), binary search for \(- (a[i] + a[j])\).

```
input: 30 -40 -20 -10 40 0 10 5
sort: -40 -20 -10 0 5 10 30 40
binary search:
(-40, -20)  60
(-40, -10)  50
(-40,  0)   40
(-40,  5)   30
(-40, 10)   20
(-40, 40)   10
(-10,  0)   10
(-20,  10)  10
( 10,  30)  -40
( 10,  40)  -30
( 30,  40)  -70
```

Analysis. Order of growth is \( N^2 \log N \).
• Step 1: \( N^2 \) with insertion sort.
• Step 2: \( N^2 \log N \) with binary search.
Comparing programs

**Hypothesis.** The $N^2 \log N$ three-sum algorithm is significantly faster in practice than the brute-force $N^3$ algorithm.

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>0.1</td>
</tr>
<tr>
<td>2.000</td>
<td>0.8</td>
</tr>
<tr>
<td>4.000</td>
<td>6.4</td>
</tr>
<tr>
<td>8.000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

ThreeSum.java

Guiding principle. Typically, better order of growth $\Rightarrow$ faster in practice.

Types of analyses

**Best case.** Lower bound on cost.
- Determined by “easiest” input.
- Provides a goal for all inputs.

**Worst case.** Upper bound on cost.
- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

**Average case.** Expected cost for random input.
- Need a model for “random” input.
- Provides a way to predict performance.

**Ex 1.** Array accesses for brute-force 3 sum.
- Best: $\sim \frac{1}{2} N^3$
- Average: $\sim \frac{1}{2} N^3$
- Worst: $\sim \frac{1}{2} N^3$

**Ex 2.** Compares for binary search.
- Best: $\sim N$
- Average: $\sim \lg N$
- Worst: $\sim \lg N$

Analysis of Algorithms

- Observations
- Mathematical models
- Order-of-growth classifications
- Dependencies on inputs
- Memory

**Types of analyses**

**Best case.** Lower bound on cost.

**Worst case.** Upper bound on cost.

**Average case.** “Expected” cost.

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.
Theory of Algorithms

Goals.
- Establish “difficulty” of a problem.
- Develop “optimal” algorithms.

Approach.
- Suppress details in analysis: analyze “to within a constant factor”.
- Eliminate variability in input model by focusing on the worst case.

Optimal algorithm.
- Performance guarantee (to within a constant factor) for any input.
- No algorithm can provide a better performance guarantee.

Commonly-used notations

<table>
<thead>
<tr>
<th>notation</th>
<th>provides example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>~ 10 N</td>
<td>10 N</td>
</tr>
<tr>
<td>Big Theta</td>
<td>asymptotic growth rate</td>
<td>Ø</td>
<td>½ N</td>
</tr>
<tr>
<td>Big Oh</td>
<td>Ø</td>
<td>Ø/N</td>
<td>10 N</td>
</tr>
<tr>
<td>Big Omega</td>
<td>Ø</td>
<td>Ω</td>
<td>½ N</td>
</tr>
</tbody>
</table>

Common mistake. Interpreting big-Oh as an approximate model.

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).

Theory of algorithms: example 1

Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = “Is there a 0 in the array?”

Upper bound. A specific algorithm.
- Running time of the optimal algorithm for 1-SUM is O(N).

Lower bound. Proof that no algorithm can do better.
- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is Ω(N).

Optimal algorithm.
- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is Θ(N).
Goals.
• Establish “difficulty” of a problem and develop “optimal” algorithms.
• Ex. 3-SUM

Upper bound. A specific algorithm.
• Ex. Brute-force algorithm for 3-SUM
• Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Goals.
• Establish “difficulty” of a problem and develop “optimal” algorithms.
• Ex. 3-SUM

Upper bound. A specific algorithm.
• Ex. Improved algorithm for 3-SUM
• Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.
• Ex. Have to examine all N entries to solve 3-SUM.
• Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.
• Optimal algorithm for 3-SUM?
• Subquadratic algorithm for 3-SUM?
• Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.
• Develop an algorithm.
• Prove a lower bound.

Gap!
• Lower the upper bound (discover a new algorithm).
• Raise the lower bound (more difficult).

Golden Age of Algorithm Design.
• 1970s-.
• Steadily decreasing upper bounds for many important problems.
• Many known optimal algorithms.

Caveats.
• Overly pessimistic to focus on worst case?
• Need better than “to within a constant factor” to predict performance.
Basics

Bit. 0 or 1.
Byte. 8 bits.
Megabyte (MB). 1 million or $2^{20}$ bytes.
Gigabyte (GB). 1 billion or $2^{30}$ bytes.

Old machine. We used to assume a 32-bit machine with 4 byte pointers.

Modern machine. We now assume a 64-bit machine with 8 byte pointers.
- Can address more memory.
- Pointers use more space.

Some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost.

Typical memory usage for objects in Java

Object overhead. 16 bytes.
Reference. 8 bytes.
Padding. Each object uses a multiple of 8 bytes.

Ex 1. A Date object uses 32 bytes of memory.

```java
public class Date
{
    private int day;
    private int month;
    private int year;
    ...
}
```

Ex 2. A String object (Java library) of length N uses \( \sim 2N \) bytes of memory.

```java
public class String
{
    private char[] value;
    private int offset;
    private int count;
    private int hash;
    ...
}
```

Typical memory usage for primitive types and arrays

<table>
<thead>
<tr>
<th>Primitive types.</th>
<th>Array overhead. 24 bytes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>bytes</td>
</tr>
<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>
</tbody>
</table>

Typical memory usage for objects in Java

Ex 2. A String and a substring:

```java
String genome = "CGCCTGGCGTCTGTAC";
String codon = genome.substring(6, 3);
```
Total memory usage for a data type value:
- Primitive type: 4 bytes for int, 8 bytes for double, …
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable + 8 if inner class.

Shallow memory usage: Don’t count referenced objects.

Deep memory usage: If array entry or instance variable is a reference, add memory (recursively) for referenced object.

Turning the crank: summary

Empirical analysis.
- Execute program to perform experiments.
- Assume power law and formulate a hypothesis for running time.
- Model enables us to make predictions.

Mathematical analysis.
- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.

Scientific method.
- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.