Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

**BHM 202 - ALGORITHMS**

**HACETEPE UNIVERSITY**
**DEPT. OF COMPUTER ENGINEERING**

**ERKUT ERDEM**

**INTRODUCTION, ANALYSIS OF ALGORITHMS**

Mar. 7, 2013

---

**INSTRUCTOR AND COURSE SCHEDULE**

- Dr. Erkut ERDEM
- erkut@cs.hacettepe.edu.tr
- Office: 114
- Tel: 297 7500 / 149

- Lectures: Thursday, 13:00-15:45@D2 (Section 2)
- Practicum (BBM204): Friday, 15:00-16:45@D4 (Section 3-5)
This course concerns programming and problem solving, with applications.

The aim is to teach students how to develop algorithms in order to solve the complex problems in the most efficient way.

The students are expected to develop a foundational understanding and knowledge of key concepts that underlie important algorithms in use on computers today.

The students are also expected to gain hands-on experience via a set of programming assignments supplied in the complementary BBM 204 Software Practicum.

About BBM202-204

Their impact is broad and far-reaching.

- Internet. Web search, packet routing, distributed file sharing, ...
- Biology. Human genome project, protein folding, ...
- Computers. Circuit layout, file system, compilers, ...
- Computer graphics. Movies, video games, virtual reality, ...
- Security. Cell phones, e-commerce, voting machines, ...
- Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...
- Social networks. Recommendations, news feeds, advertisements, ...
- Physics. N-body simulation, particle collision simulation, ...

Why study algorithms?

- Study of algorithms dates at least to Euclid.
- Formalized by Church and Turing in 1930s.
- Some important algorithms were discovered by undergraduates in a course like this!

Why study algorithms?

Old roots, new opportunities.

- Study of algorithms dates at least to Euclid.
- Formalized by Church and Turing in 1930s.
- Some important algorithms were discovered by undergraduates in a course like this!

Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity.
Why study algorithms?

For intellectual stimulation.

“For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.” — Francis Sullivan

“It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e. express it as an algorithm. The attempt to formalise things as algorithms lead to a much deeper understanding than if we simply try to comprehend things in the traditional way. Algorithm must be seen to be believed.” — Donald Knuth

For fun and profit.

Google

Apple Computers

Facebook

Cisco Systems

IBM

Nintendo

Jane Street

Morgan Stanley

Netflix

Adobe

DE Shaw & Co

Oracle

Pandora

Akamai

Yahoo!

Amazon

Microsoft

Pixar
Why study algorithms?

• Their impact is broad and far-reaching.
• Old roots, new opportunities.
• To solve problems that could not otherwise be addressed.
• For intellectual stimulation.
• To become a proficient programmer.
• They may unlock the secrets of life and of the universe.
• For fun and profit.

Why study anything else?

Communication

• The course webpage will be updated regularly throughout the semester with lecture notes, programming assignments and important deadlines. 

Getting help

• Office Hours
  - See webpage for the schedule

• BBM204 Software Practicum
  - Course related recitations, practice with algorithms, etc.

• Communication
  - Announcements and course related discussions
  - through piazza:
    BBM 202: https://piazza.com/hacettepe.edu.tr/spring2013/bbm202
    BBM 204: https://piazza.com/hacettepe.edu.tr/spring2013/bbm204

Teaching Staff

Dr. Erkut Erdem
erkut@cs.hacettepe.edu.tr
Tel: 297 7500 / 149

Dr. Burcu Can
burcucan@cs.hacettepe.edu.tr
Tel: 297 7500 / 121

Pelin Aktaş
pelinaktas@cs.hacettepe.edu.tr
Tel: 297 7500

Yasin Şahin
yasin@cs.hacettepe.edu.tr
Tel: 297 7500 / 123
Coursework and grading

Quizzes. 12% + 12%
• Two closed-book quizzes
  • in class on Thursday, April 4 and Thursday, May 23

Midterm exam. 32%
• Closed-book
  • in class on Thursday, April 25

Final exam. 40%
• Closed-book
  • Scheduled by Registrar.

Class participation. 4%
• Contribute to Piazza discussions.
  • Attend and participate in lecture.

Cheating

What is cheating?
• Sharing code: by copying, retyping, looking at, or supplying a file
• Coaching: helping your friend to write a programming assignment, line by line
• Copying code from previous course or from elsewhere on WWW

What is NOT cheating?
• Explaining how to use systems or tools
• Helping others with high-level design issues

Penalty for cheating:
• Helping others with high-level design issues
• Removal from course with failing grade

Detection of cheating:
• We do check
  • Our tools for doing this are much better than most cheaters think!

BBM204 Software Practicum

Programming assignments (PAs)
• Five assignments throughout the semester.
  • Each assignment has a well-defined goal such as solving a specific problem.
  • You must work alone on all assignments stated unless otherwise.

Important Dates
• 25 Mar. 2013, PA1
• 08 Apr. 2013, PA2
• 29 Apr. 2013, PA3
• 13 May 2013, PA4
• 27 May 2013, PA5

Resources (textbook)


Booksite.
• Brief summary of content.
• Download code from book.

http://www.algs4.princeton.edu
Course outline

Introduction
Analysis of Algorithms (2 weeks)
• Computational Complexity
• Recursion, Recurrence Relations

Sorting (1 week)
• Review
• HeapSort

Searching (3 weeks)
• Sequential Search
• Binary Search Tree
• Balanced Trees
• Hashing,
• Search Applications

Graphs (2 weeks)
• Undirected Graphs
• Directed Graphs
• Minimum Spanning Trees,
• Shortest Path

Strings (3 weeks)
• String Sorts, Tries
• Substring Search
• Regular Expressions
• Data Compression

Advanced Topics (2 weeks)
• Dynamic Programming
• Combinatorial Search, NP Completeness

Analysis of Algorithms

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

Cast of characters

**Programmer** needs to develop a working solution.

**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.

Friedrich Gauss 1805

\[time\] \[size\]

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

\[quadratic\] \[linear\] \[linearithmic\]

- 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.

Andrew Appel
PU '81

\[time\] \[size\]

<table>
<thead>
<tr>
<th>size</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>1T</td>
<td>2T</td>
<td>4T</td>
<td>8T</td>
</tr>
</tbody>
</table>

\[quadratic\] \[linear\] \[linearithmic\]
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.

Analysis of Algorithms

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

Example: 3-sum

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

30 -40 -20 -10 40 0 10 5

% java ThreeSum 8ints.txt
4

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));
    }
}
```

Measuring the running time

Q. How to time a program?

A. Manual.

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

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

Q. How to time a program?

A. Automatic.
**Empirical analysis**

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>0.0</td>
</tr>
<tr>
<td>500</td>
<td>0.0</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>?</td>
</tr>
</tbody>
</table>

**Data analysis**

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

**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.**
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>lg ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
<td>4.8</td>
<td>2.3</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>6.9</td>
<td>2.8</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>7.7</td>
<td>2.9</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>8.0</td>
<td>3.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

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

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis seems to converge to a constant $b = 3$.

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>0.11</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 ~ $c_2 N$ 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, ...

determines exponent $b$ in power law
determines constant $a$ in power law

can be compared and basic parameters identified

e.g., can run huge number of experiments

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

In practice, constant factors matter too!

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

Almost identical hypothesis to one obtained via linear regression.

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

$51.1 = a \times 8000^3$

$51.1 = a \times 0.998 \times 10^{-10}$
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.

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>integer add</td>
<td>a + b</td>
<td>2.1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>a * b</td>
<td>2.4</td>
</tr>
<tr>
<td>integer divide</td>
<td>a / b</td>
<td>5.4</td>
</tr>
<tr>
<td>floating-point add</td>
<td>a + b</td>
<td>4.6</td>
</tr>
<tr>
<td>floating-point multiply</td>
<td>a * b</td>
<td>4.2</td>
</tr>
<tr>
<td>floating-point divide</td>
<td>a / b</td>
<td>13.5</td>
</tr>
<tr>
<td>sine</td>
<td>Math.sin(theta)</td>
<td>91.3</td>
</tr>
<tr>
<td>arctangent</td>
<td>Math.atan2(y, x)</td>
<td>129.0</td>
</tr>
</tbody>
</table>

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

Novice mistake. Abusive string concatenation.
Example: 1-sum

Q. How many instructions as a function of input size $N$?

```c
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 $2N$</td>
</tr>
</tbody>
</table>

Example: 2-sum

Q. How many instructions as a function of input size $N$?

```c
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++;
```

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

Simplifying the calculations

"It is convenient to have a measure of 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 give 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

Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```c
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++;
```

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

**Ex 1.**
$$\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$$

**Ex 2.**
$$\frac{1}{6}N^3 + 100\frac{N^4}{3} + 56 \sim \frac{1}{6}N^3$$

**Ex 3.**
$$\frac{1}{6}N^3 - \frac{1}{2}N^2 + \frac{1}{3}N \sim \frac{1}{6}N^3$$

**Simplification 2: tilde notation**

discard lower-order terms
(e.g., $N = 1000$: 500 thousand vs. 166 million)

---

**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. $1 + 2 + \ldots + N$.  
$$\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x \, dx \sim \frac{1}{2} N^2$$

Ex 2. $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$.  
$$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} \, dx = \ln N$$

Ex 3. 3-sum triple loop.  
$$\sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3$$

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.
### Practical implications of order-of-growth

<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>1</td>
<td>constant</td>
<td>( a = b + c; )</td>
<td>statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td>{ while ( N &gt; 1 )  ( N = N / 2; \ldots ) }</td>
<td>divide in half</td>
<td>binary search</td>
<td>( \sim 1 )</td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td>for ( { \text{int} i = 0; i &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 ( { \text{int} i = 0; i &lt; N; i++ } )</td>
<td>double loop</td>
<td>check all pairs</td>
<td>4</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>cubic</td>
<td>for ( { \text{int} i = 0; i &lt; N; i++ } )</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>
<th>time to process millions of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>( \log N )</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>( N )</td>
<td>millions</td>
<td>tens of millions</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>hundreds of thousands</td>
<td>millions</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>hundreds</td>
<td>thousand</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>hundred</td>
<td>hundreds</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>exponential</td>
<td>useful only for tiny problems</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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>independent of input size</td>
<td>time for 100x more data</td>
</tr>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td>nearly independent of input size</td>
<td>size for 100x faster computer</td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td>optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>linearithmic</td>
<td>nearly optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>quadratic</td>
<td>not practical for large problems</td>
<td>several hours</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>cubic</td>
<td>not practical for medium problems</td>
<td>several weeks</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>exponential</td>
<td>useful only for tiny problems</td>
<td>forever</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.

<table>
<thead>
<tr>
<th>6</th>
<th>13</th>
<th>14</th>
<th>25</th>
<th>33</th>
<th>43</th>
<th>51</th>
<th>53</th>
<th>64</th>
<th>72</th>
<th>84</th>
<th>93</th>
<th>95</th>
<th>96</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

↑↑
lo
mid
hi

---

**Binary search demo**

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

**Successful search.** Binary search for 33.

<table>
<thead>
<tr>
<th>6</th>
<th>13</th>
<th>14</th>
<th>25</th>
<th>33</th>
<th>43</th>
<th>51</th>
<th>53</th>
<th>64</th>
<th>72</th>
<th>84</th>
<th>93</th>
<th>95</th>
<th>96</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

↑↑
lo
mid
hi

---

**Binary search demo**

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

**Successful search.** Binary search for 33.

<table>
<thead>
<tr>
<th>6</th>
<th>13</th>
<th>14</th>
<th>25</th>
<th>33</th>
<th>43</th>
<th>51</th>
<th>53</th>
<th>64</th>
<th>72</th>
<th>84</th>
<th>93</th>
<th>95</th>
<th>96</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

↑↑
lo
mid
hi
**Binary search demo**

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

**Successful search.** Binary search for 33.

```
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
```

- \( \text{lo} = \text{hi} \)
- \( \text{return 4} \)

**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
```

- \( \text{lo} \)
- \( \text{mid} \)
- \( \text{hi} \)
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.

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.

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 compares to search in a sorted array of size N.

Def. T(N) = # compares to binary search in a sorted subarray of size at most N.

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

Pf sketch.

T(N) ≤ T(N/2) + 1
    ≤ T(N/4) + 1 + 1
    ≤ T(N/8) + 1 + 1 + 1
    ...
    ≤ T(N/N) + 1 + 1 + ... + 1
    = 1 + lg N

given
apply recurrence to first term
apply recurrence to first term
...
stop applying, T(1) = 1
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 $-(a[i] + a[j])$.

**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.

**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:** $\sim 1$
- **Average:** $\sim \log N$
- **Worst:** $\sim \log N$
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.

Commonly-used notations

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>$\sim 10N^2$</td>
<td>$10N^2$, $10N^2 + 22N \log N$</td>
<td>provide approximate model</td>
</tr>
<tr>
<td>Big Theta</td>
<td>asymptotic</td>
<td>$O(N^2)$</td>
<td>$\frac{1}{2}N^2$, $10N^2$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td></td>
<td>growth rate</td>
<td></td>
<td>$5N^2 + 22N \log N + 3N$</td>
<td></td>
</tr>
<tr>
<td>Big Oh</td>
<td>$O(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$10N^2$, $100N$</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$22N \log N + 3N$</td>
<td></td>
</tr>
<tr>
<td>Big Omega</td>
<td>$O(N^2)$ and larger</td>
<td>$O(N^2)$</td>
<td>$\frac{1}{2}N^2$, $N^3$</td>
<td>develop lower bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$N^3 + 22N \log N + 3N$</td>
<td></td>
</tr>
</tbody>
</table>

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

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.

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 $\Omega(N)$.

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 $\Theta(N)$.

**Theory of algorithms: example 2**

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 $O(N^3)$.

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 $\Omega(N)$.

Optimal algorithm.
- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 3-SUM is optimal: its running time is $\Theta(N)$.

**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.
## Analysis of Algorithms

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

### Basics

<table>
<thead>
<tr>
<th>Bit</th>
<th>0 or 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8 bits</td>
</tr>
<tr>
<td>Megabyte (MB)</td>
<td>1 million or $2^{20}$ bytes</td>
</tr>
<tr>
<td>Gigabyte (GB)</td>
<td>1 billion or $2^{30}$ bytes</td>
</tr>
</tbody>
</table>

**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 primitive types and objects in Java

#### Primitive types.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

**Array overhead.** 24 bytes.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>$2N + 24$</td>
</tr>
<tr>
<td>int[]</td>
<td>$4N + 24$</td>
</tr>
<tr>
<td>double[]</td>
<td>$8N + 24$</td>
</tr>
</tbody>
</table>

For one-dimensional arrays.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[][]</td>
<td>$\sim 2MN$</td>
</tr>
<tr>
<td>int[][]</td>
<td>$\sim 4MN$</td>
</tr>
<tr>
<td>double[][]</td>
<td>$\sim 8MN$</td>
</tr>
</tbody>
</table>

For two-dimensional arrays.

#### 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;
    ...
}
```

<table>
<thead>
<tr>
<th>object overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>day</td>
</tr>
<tr>
<td>month</td>
</tr>
<tr>
<td>year</td>
</tr>
<tr>
<td>padding</td>
</tr>
</tbody>
</table>

16 bytes (object overhead)
- 4 bytes (int)
- 4 bytes (int)
- 4 bytes (int)
- 4 bytes (padding)
- 32 bytes
Typical memory usage for objects in Java

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

Ex 2. A virgin string of length \( N \) uses \( \sim 2N \) bytes of memory.

public class String
  {
    private char[] value;
    private int offset;
    private int count;
    private int hash;
    ...
  }  
16 bytes (object overhead)
8 bytes (reference to array)
2N + 24 bytes (char[] array)
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)
2N + 64 bytes

Typical memory usage summary

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.

Memory profiler

Classmexer library. Measure memory usage of a Java object by querying JVM.

http://www.javamex.com/classmexer

import com.javamex.classmexer.MemoryUtil;

public class Memory {
  public static void main(String[] args) {
    Date date = new Date(12, 31, 1999);
    StdOut.println(MemoryUtil.memoryUsageOf(date));
    String s = "Hello, World";
    StdOut.println(MemoryUtil.memoryUsageOf(s));
    StdOut.println(MemoryUtil.deepMemoryUsageOf(s));
  }
}

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.