1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory
1.4 **Analysis of Algorithms**

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory
Cast of characters

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

**Client** wants to solve problem efficiently.

**Theoretician** wants to understand.

**Student** might play any or all of these roles someday.
“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 (COS 226)*
- Provide guarantees.
- Understand theoretical basis.  
  
  *theory of algorithms (COS 423)*

**Primary practical reason:** avoid performance bugs.

*client gets poor performance because programmer did not understand performance characteristics*
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.
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.**
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?

**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.
1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory
Example: 3-SUM

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

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5
% java ThreeSum 8ints.txt
4
```

```
<table>
<thead>
<tr>
<th>a[i]</th>
<th>a[j]</th>
<th>a[k]</th>
<th>sum</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) {
        In in = new In(args[0]);
        int[] a = in.readAllInts();
        StdOut.println(count(a));
    }
}
```
Measuring the running time

Q. How to time a program?
A. Manual.
# Measuring the running time

**Q.** How to time a program?

**A.** Automatic.

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

    Stopwatch()  
        create a new stopwatch

    double elapsedTime()
        time since creation (in seconds)

public static void main(String[] args)
{
    In in = new In(args[0]);
    int[] a = in.readInts();
    Stopwatch stopwatch = new Stopwatch();
    StdOut.println(ThreeSum.count(a));
    double time = stopwatch.elapsedTime();
    StdOut.println("elapsed time " + time);
}
```
Empirical analysis

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

![Standard plot diagram](image-url)
Data analysis

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

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

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.
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>51.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>410.8</td>
</tr>
</tbody>
</table>

"order of growth" of running time is about $N^3$ [stay tuned]

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>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>

\[
\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = 2^b
\]

\[
\lg (6.4 / 0.8) = 3.0
\]

**Hypothesis.** Running time is about $a N^b$ with $b = \lg \text{ratio}$.

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

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

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) $\uparrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>8,000</td>
<td>51.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

$51.1 = a \times 8000^3$

$\Rightarrow a = 0.998 \times 10^{-10}$

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

almost identical hypothesis to one obtained via linear regression
Experimental algorithmics

System independent effects.
• Algorithm.
• Input data.

\[ \{ \text{determines exponent } b \text{ in power law} \} \]

System dependent effects.
• Hardware: CPU, memory, cache, ...
• Software: compiler, interpreter, garbage collector, ...
• System: operating system, network, other apps, ...

\[ \{ \text{determines constant } a \text{ in power law} \} \]

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

\[ \text{e.g., can run huge number of experiments} \]
1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- 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

**Challenge.** How to estimate constants.

<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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM
Cost of basic operations

Observation. Most primitive operations take constant time.

<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>$c_1$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>a = b</td>
<td>$c_2$</td>
</tr>
<tr>
<td>integer compare</td>
<td>a &lt; b</td>
<td>$c_3$</td>
</tr>
<tr>
<td>array element access</td>
<td>a[i]</td>
<td>$c_4$</td>
</tr>
<tr>
<td>array length</td>
<td>a.length</td>
<td>$c_5$</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[N]</td>
<td>$c_6 N$</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[N][N]</td>
<td>$c_7 N^2$</td>
</tr>
</tbody>
</table>

Caveat. Non-primitive operations often take more than constant time.

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 $N$?

```java
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++;
```

Pf. [ n even]

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

$$= \binom{N}{2}$$

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

half of square

half of diagonal
String theory infinite sum

\[ 1 + 2 + 3 + 4 + \ldots = -\frac{1}{12} \]

http://www.nytimes.com/2014/02/04/science/in-the-end-it-all-adds-up-to.html
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++;

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

<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>

tedious to count exactly
Simplifying the calculations

“\textit{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 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

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

\textbf{By A. M. TURING}

\textit{(National Physical Laboratory, Teddington, Middlesex)}

[Received 4 November 1947]

\textbf{SUMMARY}

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known ‘Gauss elimination process’, it is found that the errors are normally quite moderate; no exponential build-up need occur.
**Simplification 1: cost model**

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

```java
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++;
```

Simplification: 0 + 1 + 2 + \ldots + (N - 1) = \frac{1}{2} N (N - 1)

<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><strong>array access</strong></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>

(cost model = array accesses)

(we assume compiler/JVM do not optimize any array accesses away!)
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. $\frac{1}{6} N^3 + 20 N + 16 \sim \frac{1}{6} N^3$

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

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

discard lower-order terms (e.g., $N = 1000$: 166.67 million vs. 166.17 million)

Technical definition. $f(N) \sim g(N)$ 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>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
<td>$\sim N^2$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$ to $\sim N^2$</td>
</tr>
</tbody>
</table>
Example: 2-SUM

Q. Approximately how many array accesses as a function of input size $N$?

```java
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++;
```

A. $\sim N^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.
**Example: 3-SUM**

**Q.** Approximately how many array accesses as a function of input size $N$?

```java
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++;
```

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

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

**Bottom line.** Use cost model and tilde notation to simplify counts.
Diversion: estimating a discrete sum

Q. How to estimate a discrete sum?
A1. Take a discrete mathematics course (COS 340).
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. $1^k + 2^k + \ldots + N^k.$

$$\sum_{i=1}^{N} i^k \sim \int_{x=1}^{N} x^k \, dx \sim \frac{1}{k+1} N^{k+1}$$

Ex 3. $1 + 1/2 + 1/3 + \ldots + 1/N.$

$$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} \, dx = \ln N$$

Ex 4. 3-sum triple loop.

$$\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3$$
Estimating a discrete sum

Q. How to estimate a discrete sum?
A1. Take a discrete mathematics course (COS 340).
A2. Replace the sum with an integral, and use calculus!

Ex 4. \(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\)

\[
\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2
\]

\[
\int_{x=0}^{\infty} \left(\frac{1}{2}\right)^x \, dx = \frac{1}{\ln 2} \approx 1.4427
\]

Caveat. Integral trick doesn't always work!
Estimating a discrete sum

Q. How to estimate a discrete sum?
A. Use Maple or Wolfram Alpha.
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.

Mathematical models for running time

\[ T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E \]

- \( A = \) array access
- \( B = \) integer add
- \( C = \) integer compare
- \( D = \) increment
- \( E = \) variable assignment

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

How many array accesses does the following code fragment make as a function of $N$?

```c
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = 1; k < N; k = k*2)
            if (a[i] + a[j] >= a[k])
                count++;
```

A. $\sim 3 \ N^2$
B. $\sim 3/2 \ N^2 \ lg \ N$
C. $\sim 3/2 \ N^3$
D. $\sim 3 \ N^3$
E. I don't know.
1.4 **Analysis of Algorithms**

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory
Common order-of-growth classifications

**Definition.** If $f(N) \sim c g(N)$ for some constant $c > 0$, then the order of growth of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

**Ex.** The order of growth of the *running time* of this code is $N^3$.

```java
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++;
```

**Typical usage.** With running times.

where leading coefficient depends on machine, compiler, JVM, ...
**Good news.** The set of functions

$$1, \; \log N, \; N, \; N \log N, \; N^2, \; N^3, \; \text{and} \; 2^N$$

suffices to describe the order of growth of most common algorithms.
## Common order-of-growth classifications

<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>
</tbody>
</table>
| log $N$         | logarithmic    | while $(N > 1)$ 
{ 
    $N = N/2$; ... 
} | divide in half | binary search        | $\sim 1$      |
| $N$             | linear         | for $(\text{int } i = 0; i < N; i++)$ 
{ 
    ... 
} | loop          | find the maximum     | 2             |
| $N \log N$     | linearithmic   | [see mergesort lecture] | divide and conquer | mergesort | $\sim 2$ |
| $N^2$           | quadratic      | for $(\text{int } i = 0; i < N; i++)$ 
for $(\text{int } j = 0; j < N; j++)$ 
{ 
    ... 
} | double loop    | check all pairs     | 4             |
| $N^3$           | cubic          | for $(\text{int } i = 0; i < N; i++)$ 
for $(\text{int } j = 0; j < N; j++)$ 
for $(\text{int } k = 0; k < N; k++)$ 
{ 
    ... 
} | triple loop    | check all triples   | 8             |
| $2^N$           | exponential    | [see combinatorial search lecture] | exhaustive search | check all subsets | $T(N)$        |
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.
Binary search: implementation

Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's Arrays.binarySearch() discovered in 2006.

Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Posted by Joshua Bloch, Software Engineer

I remember vividly Jon Bentley’s first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful *Programming Pearls* (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

http://googleresearch.blogspot.com/2006/06 EXTRA EXTRA READ ALL ABOUT IT NEARLY.html
Invariant. If key appears in array $a[]$, then $a[lo] \leq key \leq a[hi]$.

```java
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;
}
```

why not mid = (lo + hi) / 2 ?

one "3-way compare"
Binary search: mathematical analysis

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

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

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

Pf sketch. [assume $N$ is a power of 2]

\[
\begin{align*}
T(N) & \leq T(N/2) + 1 \quad \text{[ given ]} \\
& \leq T(N/4) + 1 + 1 \quad \text{[ apply recurrence to first term ]} \\
& \leq T(N/8) + 1 + 1 + 1 \quad \text{[ apply recurrence to first term ]} \\
& \vdots \\
& \leq T(N/N) + 1 + 1 + \ldots + 1 \quad \text{[ stop applying, } T(1) = 1 \text{ ]} \\
& = 1 + \lg N
\end{align*}
\]
TECHNICAL INTERVIEW QUESTIONS
WHY ARE MANHOLE COVERS ROUND?
**The 3-Sum Problem**

**3-Sum.** Given $N$ distinct integers, find three such that $a + b + c = 0$.

**Version 0.** $N^3$ time, $N$ space.
**Version 1.** $N^2 \log N$ time, $N$ space.
**Version 2.** $N^2$ time, $N$ space.

**Note.** For full credit, running time should be worst case.
An $N^2 \log N$ algorithm for 3-SUM

**Algorithm.**
- Step 1: Sort the $N$ (distinct) numbers.
- Step 2: 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.

**Remark.** Can achieve $N^2$ by modifying binary search step.

- **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) |  35   |
  |   (-40,       10)|  30   |
  |                  |   ... |
  |   (-20,       -10)|  30   |
  |                  |   ... |       |
  |   (-10,       0)  |  10   |
  |                  |   ... |       |
  |    (   10,      30)| -40   |
  |    (   10,      40)| -50   |
  |    (   30,      40)| -70   |

- only count if $a[i] < a[j] < a[k]$ to avoid double counting
Comparing programs

**Hypothesis.** The sorting-based $N^2 \log N$ algorithm for 3-SUM 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*

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.14</td>
</tr>
<tr>
<td>2,000</td>
<td>0.18</td>
</tr>
<tr>
<td>4,000</td>
<td>0.34</td>
</tr>
<tr>
<td>8,000</td>
<td>0.96</td>
</tr>
<tr>
<td>16,000</td>
<td>3.67</td>
</tr>
<tr>
<td>32,000</td>
<td>14.88</td>
</tr>
<tr>
<td>64,000</td>
<td>59.16</td>
</tr>
</tbody>
</table>

*ThreeSumDeluxe.java*

**Guiding principle.** Typically, better order of growth $\Rightarrow$ faster in practice.
1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory
Basics

Bit. 0 or 1.  

Byte. 8 bits.

Megabyte (MB). 1 million or $2^{20}$ bytes.

Gigabyte (GB). 1 billion or $2^{30}$ bytes.

64-bit machine. We assume a 64-bit machine with 8-byte pointers.

- Can address more memory.
- Pointers use more space.

\[ \text{some JVMs "compress" ordinary object} \]
\[ \text{pointers to 4 bytes to avoid this cost} \]
### Typical memory usage for primitive types and arrays

<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>

**primitive types**

<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>

**one-dimensional arrays**

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[][]</td>
<td>~2MN</td>
</tr>
<tr>
<td>int[][]</td>
<td>~4MN</td>
</tr>
<tr>
<td>double[][]</td>
<td>~8MN</td>
</tr>
</tbody>
</table>

**two-dimensional arrays**
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;
    ...
}
```

16 bytes (object overhead)

4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)
32 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.
- Padding: round up to multiple of 8 bytes.

Note. Depending on application, we may want to count memory for any referenced objects (recursively).
Memory analysis quiz

How much memory does a WeightedQuickUnionUF use as a function of $N$?

A. $\sim 4N$ bytes
B. $\sim 8N$ bytes
C. $\sim 4N^2$ bytes
D. $\sim 8N^2$ bytes
E. I don't know

```java
public class WeightedQuickUnionUF {
    private int[] id;
    private int[] sz;
    private int count;

    public WeightedQuickUnionUF(int N) {
        id = new int[N];
        sz = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
        for (int i = 0; i < N; i++) sz[i] = 1;
        ...
    }
}
```
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.