1.4 Analysis of Algorithms

- introduction
- running time (experimental analysis)
- running time (mathematical models)
- order-of-growth classifications
- memory usage

see precept 1
1.4 Analysis of Algorithms

- introduction
- running time (experimental analysis)
- running time (mathematical models)
- order-of-growth classifications
- memory usage
Cast of characters

Programmer needs to develop a working solution.

Client wants to solve problem efficiently.

Theoretician seeks to understand.

Student (you) might play any or all of these roles someday.
Running time

“As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then 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.

Provide guarantees.

Understand theoretical basis.

Primary practical reason: avoid performance bugs.

client gets poor performance because programmer did not understand performance characteristics
An algorithmic success story

N-body simulation.
- Simulate gravitational interactions among $n$ bodies.
- Applications: cosmology, fluid dynamics, semiconductors, ...
- 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?

Our approach. Combination of experiments and mathematical modeling.
1.4 **Analysis of Algorithms**

- introduction
- *running time (experimental analysis)*
- *running time (mathematical models)*
- order-of-growth classifications
- memory usage
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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a[i]</td>
<td>a[j]</td>
<td>a[k]</td>
<td>sum</td>
</tr>
<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.** Related to problems in computational geometry.
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 algs4.jar)
    public Stopwatch() {  // create a new stopwatch
        ...
    }
    public double elapsedTime() {  // time since creation (in seconds)
        ...
    }
}

client code

public static void main(String[] args) {
    In in = new In(args[0]);
    int[] a = in.readInts();
    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</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>?</td>
</tr>
</tbody>
</table>

† on a 2.8GHz Intel PU-226 with 64GB DDR E3 memory and 32MB L3 cache; running Oracle Java 1.7.0_45-b18 on Springdale Linux v. 6.5
Data analysis

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

**Hypothesis (power law).** $T(n) = a n^b$.

**Questions.** How to validate hypothesis? How to estimate $a$ and $b$?
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.

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

$$\log_2(T(n)) = 2.999 \log_2 n + (-33.2103)$$

$$T(n) = 2^{-33.2103} \times n^{2.999}$$
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) $^\dagger$</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>

“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</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>500</td>
<td>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</td>
<td>3</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\frac{T(n)}{T(n/2)} = \frac{an^b}{a(n/2)^b} = 2^b
\]

\[\log_2 \left( \frac{6.4}{0.8} \right) = 3.0\]

seems to converge to a constant $b \approx 3$

**Hypothesis.** Running time is about $an^b$ with $b = \log_2$ 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) †</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>

\[ 51.1 = a \times 8000^3 \]

\[ \Rightarrow a = 0.998 \times 10^{-10} \]

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

almost identical hypothesis
to one obtained via regression
Estimate the running time to solve a problem of size \( n = 96,000 \).

A. 39 seconds
B. 52 seconds
C. 117 seconds
D. 350 seconds

<table>
<thead>
<tr>
<th>( n )</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.02</td>
</tr>
<tr>
<td>2,000</td>
<td>0.05</td>
</tr>
<tr>
<td>4,000</td>
<td>0.20</td>
</tr>
<tr>
<td>8,000</td>
<td>0.81</td>
</tr>
<tr>
<td>16,000</td>
<td>3.25</td>
</tr>
<tr>
<td>32,000</td>
<td>13.01</td>
</tr>
</tbody>
</table>
Experimental algorithmics

System independent effects.
- Algorithm.
- Input data.

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

System dependent effects.
- Hardware: CPU, memory, cache, ... \[ \text{determines constant } a \]
- Software: compiler, interpreter, garbage collector, ... \[ \text{in power law } a n^b \]
- System: operating system, network, other apps, ...

![Intel CORE i9 X-series](image1)

![Java Oracle](image2)

![Ubuntu 14.04 LTS](image3)

**Bad news.** Sometimes difficult to get precise measurements.

**Good news.** Much easier and cheaper than other sciences.
An aside

Algorithmic experiments are virtually free by comparison with other sciences.

Bottom line. No excuse for not running experiments to understand costs.
Scientific method applied to the 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
- *running time (experimental analysis)*
- *running time (mathematical models)*
- order-of-growth classifications
- memory usage

https://algs4.cs.princeton.edu
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.
Example: 1-SUM

Q. How many operations 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>cost (ns) †</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2/5</td>
<td>2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>1/5</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/5</td>
<td>$n + 1$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/10</td>
<td>$n$</td>
</tr>
<tr>
<td>array access</td>
<td>1/10</td>
<td>$n$</td>
</tr>
<tr>
<td>increment</td>
<td>1/10</td>
<td>$n$ to $2n$</td>
</tr>
</tbody>
</table>

† representative estimates (with some poetic license)

in practice, depends on caching, bounds checking, ... (see COS 217)

exactly $n$ array accesses
How many array accesses as a function of n?

A. $\frac{1}{2} n (n - 1)$

B. $n (n - 1)$

C. $2 n^2$

D. No idea.

```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++;
```
Example: 2-SUM

Q. How many operations 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++;
```

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

$= \binom{n}{2}$

<table>
<thead>
<tr>
<th>operation</th>
<th>cost (ns)</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2/5</td>
<td>$n + 2$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>1/5</td>
<td>$n + 2$</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/5</td>
<td>$\frac{1}{2} (n + 1) (n + 2)$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/10</td>
<td>$\frac{1}{2} n (n - 1)$</td>
</tr>
<tr>
<td>array access</td>
<td>1/10</td>
<td>$n (n - 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>1/10</td>
<td>$\frac{1}{2} n (n + 1)$ to $n^2$</td>
</tr>
</tbody>
</table>
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++;
```

\[
0 + 1 + 2 + \ldots + (n - 1) = \frac{1}{2} n(n - 1)
\]

\[
= \binom{n}{2}
\]

<table>
<thead>
<tr>
<th>operation</th>
<th>cost (ns)</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2/5</td>
<td>( n + 2 )</td>
</tr>
<tr>
<td>assignment statement</td>
<td>1/5</td>
<td>( n + 2 )</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/5</td>
<td>( \frac{1}{2} (n + 1) (n + 2) )</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/10</td>
<td>( \frac{1}{2} n (n - 1) )</td>
</tr>
<tr>
<td>array access</td>
<td>1/10</td>
<td>( n (n - 1) )</td>
</tr>
<tr>
<td>increment</td>
<td>1/10</td>
<td>( \frac{1}{2} n (n + 1) ) to ( n^2 )</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.

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., $n = 1,000$: 166.67 million vs. 166.17 million)

Rationale.
- When $n$ is large, lower-order terms are negligible.
- When $n$ is small, we don’t care.

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.

<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 $n^2$</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.

\[
0 + 1 + 2 + \ldots + (n-1) = \frac{1}{2} n(n-1) = \binom{n}{2}
\]
Example: 3-SUM

Q. Approximately how many array accesses 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++)
        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.

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

Bottom line. Use cost model and tilde notation to simplify counts.
Estimating a discrete sum

Q. How to estimate a discrete sum?
A1. Take a discrete mathematics course (COS 340).
Estimating a discrete sum

Q. How to estimate a discrete sum?
A2. Replace the sum with an integral; use calculus!

Ex 1. \[ \sum_{i=1}^{n} i \sim \int_{x=1}^{n} x \, dx \sim \frac{1}{2} n^2 \]

Ex 2. \[ \sum_{i=1}^{n} \frac{1}{i} \sim \int_{x=1}^{n} \frac{1}{x} \, dx \sim \ln n \]

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

Ex 4. \[ \int_{x=0}^{\infty} \left( \frac{1}{2} \right)^x \, dx = \frac{1}{\ln 2} \approx 1.4427 \]
\[ \sum_{i=0}^{\infty} \left( \frac{1}{2} \right)^i = 2 \]

integral trick doesn’t always work!
Estimating a discrete sum

Q. How to estimate a discrete sum?
A3. Use Maple or Wolfram Alpha.
Analysis of algorithms: quiz 3

How many array accesses as a function of $n$?

$$\begin{align*}
\text{int count} &= 0; \\
\text{for (int i = 0; i < n; i++)} & \quad \text{for (int j = i+1; j < n; j++)} \\
& \quad \quad \text{for (int k = 1; k < n; k = k*2)} \\
& \quad \quad \quad \text{if (a[i] + a[j] >= a[k])} \\
& \quad \quad \quad \quad \text{count++;} \\
\end{align*}$$

A. $\sim n^2 \log_2 n$

B. $\sim \frac{3}{2} n^2 \log_2 n$

C. $\sim \frac{1}{2} n^3$

D. $\sim \frac{3}{2} n^3$
1.4 Analysis of Algorithms

- introduction
- running time (experimental analysis)
- running time (mathematical models)
- order-of-growth classifications
- memory usage
**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.** Mathematical analysis of 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 (int $i = 0$; $i < n$; $i++$)  
{  
  ...  
}                                      | single loop       | find the maximum      | 2              |
| $n \log n$      | linearithmic    | see mergesort lecture                                      | divide and conquer| mergesort             | $\sim 2$        |
| $n^2$           | quadratic       | for (int $i = 0$; $i < n$; $i++$)  
for (int $j = 0$; $j < n$; $j++$)  
{  
  ...  
}                                      | double loop       | check all pairs       | 4              |
| $n^3$           | cubic           | for (int $i = 0$; $i < n$; $i++$)  
for (int $j = 0$; $j < n$; $j++$)  
for (int $k = 0$; $k < n$; $k++$)  
{  
  ...  
}                                      | triple loop       | check all triples     | 8              |
| $2^n$           | exponential     | see combinatorial search lecture                           | exhaustive search | check all subsets     | $2^n$          |
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
Friday, June 02, 2006

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] ≤ key ≤ 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 $1 + \log_2 n$ key compares to search in a sorted array of length $n$.

**Def.** $T(n) = \text{max # key compares to search a sorted subarray of length } \leq n$.

**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]

\[
T(n) \leq T(n / 2) + 1 \quad [\text{given}]
\]
\[
\leq T(n / 4) + 1 + 1 \quad [\text{apply recurrence to first term}]
\]
\[
\leq T(n / 8) + 1 + 1 + 1 \quad [\text{apply recurrence to first term}]
\]
\[
\vdots
\]
\[
\leq T(n / n) + 1 + 1 + \ldots + 1 \quad [\text{stop applying, } T(1) = 1]
\]
\[
= 1 + \log_2 n
\]
Why are sewer access covers round?

New York, New York
Okayama, Japan
Zermatt, Switzerland
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, the running time should be in the worst case.
1.4 Analysis of Algorithms

- introduction
- running time (experimental analysis)
- running time (mathematical models)
- order-of-growth classifications
- memory usage
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.
## 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>boolean[]</td>
<td>$1n + 24$</td>
</tr>
<tr>
<td>int[]</td>
<td>$4n + 24$</td>
</tr>
<tr>
<td>double[]</td>
<td>$8n + 24$</td>
</tr>
</tbody>
</table>

**One-dimensional array (length n)**

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean[][]</td>
<td>$\sim 1 m n$</td>
</tr>
<tr>
<td>int[][]</td>
<td>$\sim 4 m n$</td>
</tr>
<tr>
<td>double[][]</td>
<td>$\sim 8 m n$</td>
</tr>
</tbody>
</table>

**Two-dimensional array (m-by-n)**

- wasteful
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 often want to count the memory for any referenced objects (recursively).

“deep memory”

+ 8 extra bytes per inner class object (for reference to enclosing class)
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

```java
public class WeightedQuickUnionUF {
    private int[] parent;
    private int[] size;
    private int count;

    public WeightedQuickUnionUF(int n) {
        parent = new int[n];
        size = new int[n];

        count = 0;
        for (int i = 0; i < n; i++)
            parent[i] = i;
        for (int i = 0; i < n; i++)
            size[i] = 1;
    }
    ...
}
```
**Turning the crank: summary**

**Empirical analysis.**
- Execute program to perform experiments.
- Assume power law.
- 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.

\[
\sum_{h=0}^{[\lg n]} \left\lfloor \frac{n}{2^h+1} \right\rfloor h \sim n
\]