Data Structures and Algorithms
Analysis of Algorithms

Chris Brooks

Department of Computer Science
University of San Francisco
When is algorithm A better than algorithm B?
When is algorithm A better than algorithm B?

- Algorithm A runs faster
When is algorithm A better than algorithm B?

- Algorithm A runs faster
- Algorithm A requires less space to run
When is algorithm A better than algorithm B?

- Algorithm A runs faster
- Algorithm A requires less space to run

Space / Time Trade-off

- Can often create an algorithm that runs faster, by using more space

For now, we will concentrate on time efficiency
How long does the following function take to run:

```java
boolean find(int A[], int element) {
    for (i=0; i<A.length; i++) {
        if (A[i] == elem)
            return true;
    }
    return false;
}
```
3-5: Best Case vs. Worst Case

How long does the following function take to run:

```java
boolean find(int A[], int element) {
    for (i=0; i<A.length; i++) {
        if (A[i] == elem)
            return true;
    }
    return false;
}
```

It depends on if – and where – the element is in the list.
Best Case – What is the fastest that the algorithm can run

Worst Case – What is the slowest that the algorithm can run

Average Case – How long, on average, does the algorithm take to run

Worst Case performance is almost always important. Usually, Best Case performance is unimportant (why?)

Usually, Average Case = Worst Case (but not always!)
How long does an algorithm take to run?
3-8: Measuring Time Efficiency

How long does an algorithm take to run?

- Solution 1: Implement on a computer, set a timer.
3-9: Measuring Time Efficiency

How long does an algorithm take to run?

Solution 1: Implement on a computer, set a timer.

Problems:

- Not just testing algorithm – testing implementation of algorithm
  - Different implementations might be more or less efficient

- Implementation details (cache performance, other programs running in the background, etc) can affect results

- Hardware characteristics can affect performance

- Hard to compare algorithms that are not tested under exactly the same conditions
3-10: Measuring Time Efficiency

How long does an algorithm take to run?

Solution 1: Implement on a computer, set a timer.
Problems:
△ Not just testing algorithm – testing implementation of algorithm
△ Implementation details (cache performance, other programs running in the background, etc) can affect results
△ Hard to compare algorithms that are not tested under exactly the same conditions

Better Solution: Build a mathematical model of the running time, use model to compare algorithms
3-11: Competing Algorithms

6 Linear Search

for (i=low; i <= high; i++)
  if (A[i] == elem) return true;
return false;

6 Binary Search

int BinarySearch(int low, int high, elem) {
  if (low > high) return false;
  mid = (high + low) / 2;
  if (A[mid] == elem) return true;
  if (A[mid] < elem)
    return BinarySearch(mid+1, high, elem);
  else
    return BinarySearch(low, mid-1, elem);
}
3-12: Linear vs Binary

- Linear Search
- Assume each comparison takes $c_1$ time.

Time Required, for a problem of size $n$:
Linear Search

Assume each comparison takes $c_1$ time.

Time Required, for a problem of size $n$:

$c_1 \cdot n$ for some constant $c_1$

Binary Search

Each comparison takes $c_2$ time.

Time Required, for a problem of size $n$:
Linear Search

Assume each comparison takes $c_1$ time.

Time Required, for a problem of size $n$:

$c_1 \times n$ for some constant $c_1$

Binary Search

Each comparison takes $c_2$ time.

Time Required, for a problem of size $n$:

$c_2 \times \log(n)$ for some constant $c_2$
3-15: Do Constants Matter?

- Linear Search requires time $c_1 \times n$, for some $c_1$
- Binary Search requires time $c_2 \times \log(n)$, for some $c_2$

What if there is a very high overhead cost for function calls?

What if $c_2$ is 1000 times larger than $c_1$?
### 3-16: Constants Do Not Matter!

<table>
<thead>
<tr>
<th>Length of list</th>
<th>Time Required for Linear Search</th>
<th>Time Required for Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.001 seconds</td>
<td>0.3 seconds</td>
</tr>
<tr>
<td>100</td>
<td>0.01 seconds</td>
<td>0.66 seconds</td>
</tr>
<tr>
<td>1000</td>
<td>0.1 seconds</td>
<td>1.0 seconds</td>
</tr>
<tr>
<td>10000</td>
<td>1 second</td>
<td>1.3 seconds</td>
</tr>
<tr>
<td>100000</td>
<td>10 seconds</td>
<td>1.7 seconds</td>
</tr>
<tr>
<td>10000000</td>
<td>2 minutes</td>
<td>2.0 seconds</td>
</tr>
<tr>
<td>100000000</td>
<td>17 minutes</td>
<td>2.3 seconds</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>11 days</td>
<td>3.3 seconds</td>
</tr>
<tr>
<td>$10^{15}$</td>
<td>30 centuries</td>
<td>5.0 seconds</td>
</tr>
<tr>
<td>$10^{20}$</td>
<td>300 million years</td>
<td>6.6 seconds</td>
</tr>
</tbody>
</table>
We care about the *Growth Rate* of a function – how much more we can do if we add more processing power

Faster Computers $\neq$ Solving Problems Faster
Faster Computers $= \text{Solving Larger Problems}$

- Modeling more variables
- Handling bigger databases
- Pushing more polygons
### Growth Rate Examples

<table>
<thead>
<tr>
<th>Time</th>
<th>$10n$</th>
<th>$5n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 s</td>
<td>1000</td>
<td>2000</td>
<td>1003</td>
<td>100</td>
<td>21</td>
<td>13</td>
</tr>
<tr>
<td>2 s</td>
<td>2000</td>
<td>4000</td>
<td>1843</td>
<td>141</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>20 s</td>
<td>20000</td>
<td>40000</td>
<td>14470</td>
<td>447</td>
<td>58</td>
<td>17</td>
</tr>
<tr>
<td>1 m</td>
<td>60000</td>
<td>120000</td>
<td>39311</td>
<td>774</td>
<td>84</td>
<td>19</td>
</tr>
<tr>
<td>1 hr</td>
<td>3600000</td>
<td>7200000</td>
<td>1736782</td>
<td>18973</td>
<td>331</td>
<td>28</td>
</tr>
</tbody>
</table>
3-19: Constants and Running Times

6 When calculating a formula for the running time of an algorithm:
   ▲ Constants aren’t as important as the growth rate of the function
   ▲ Lower order terms don’t have much of an impact on the growth rate
      - $x^3 + x$ vs $x^3$

6 We’d like a formal method for describing what is important when analyzing running time, and what is not.
But what about real-world performance? Aren’t we ignoring all that?
But what about real-world performance? Aren’t we ignoring all that?

Yes.

Algorithmic analysis is just the first step in designing an efficient program.

Once you’ve chosen the right algorithm, you’ll also want to instrument your code to find bottlenecks.

- Software engineering deals with this (among other issues).

But if you don’t choose the right algorithm, there’s no point in optimizing.
3-22: **Big-Oh Notation**

$O(f(n))$ is the set of all functions that are bound from above by $f(n)$

$T(n) \in O(f(n))$ if

$$\exists c, n_0 \text{ such that } T(n) \leq c \cdot f(n) \text{ when } n > n_0$$
3-23: Big-Oh Examples

\[ n \in O(n) \ ? \]
\[ 10n \in O(n) \ ? \]
\[ n \in O(10n) \ ? \]
\[ n \in O(n^2) \ ? \]
\[ n^2 \in O(n) \ ? \]
\[ 10n^2 \in O(n^2) \ ? \]
\[ n \lg n \in O(n^2) \ ? \]
\[ \ln n \in O(2n) \ ? \]
\[ \lg n \in O(n) \ ? \]
\[ 3n + 4 \in O(n) \ ? \]
\[ 5n^2 + 10n - 2 \in O(n^3) \ ? \ O(n^2) \ ? \ O(n) \ ? \]
3-24: **Big-Oh Examples**

\[ n \in O(n) \]
\[ 10n \in O(n) \]
\[ n \in O(10n) \]
\[ n \in O(n^2) \]
\[ n^2 \notin O(n) \]
\[ 10n^2 \in O(n^2) \]
\[ n \ln n \in O(n^2) \]
\[ \ln n \in O(2n) \]
\[ \lg n \in O(n) \]
\[ 3n + 4 \in O(n) \]
\[ 5n^2 + 10n - 2 \in O(n^3), \in O(n^2), \notin O(n) \]
3-25: Big-Oh Examples II

\[
\sqrt{n} \in O(n) \ ? \\
lg n \in O(2^n) \ ? \\
lg n \in O(n) \ ? \\
n\lg n \in O(n) \ ? \\
n\lg n \in O(n^2) \ ? \\
\sqrt{n} \in O(lg n) \ ? \\
lg n \in O(\sqrt{n}) \ ? \\
n\lg n \in O(n^{\frac{3}{2}}) \ ? \\
n^3 + n\lg n + n\sqrt{n} \in O(n\lg n) \ ? \\
n^3 + n\lg n + n\sqrt{n} \in O(n^3) \ ? \\
n^3 + n\lg n + n\sqrt{n} \in O(n^4) \ ?
\]
3-26: Big-Oh Examples II

\[
\sqrt{n} \in O(n) \\
lg n \in O(2^n) \\
lg n \in O(n) \\
n \lg n \not\in O(n) \\
n \lg n \in O(n^2) \\
\sqrt{n} \not\in O(lg n) \\
lg n \in O(\sqrt{n}) \\
n \lg n \in O(n^{3/2}) \\
n^3 + n \lg n + n \sqrt{n} \not\in O(n \lg n) \\
n^3 + n \lg n + n \sqrt{n} \in O(n^3) \\
n^3 + n \lg n + n \sqrt{n} \in O(n^4)
\]
3-27: Big-Oh Examples III

\[ f(n) = \begin{cases} 
  n & \text{for } n \text{ odd} \\
  n^3 & \text{for } n \text{ even} 
\end{cases} \]

\[ g(n) = n^2 \]

\[ f(n) \in O(g(n)) \] ?

\[ g(n) \in O(f(n)) \] ?

\[ n \in O(f(n)) \] ?

\[ f(n) \in O(n^3) \] ?
\[ f(n) = \begin{cases} 
  n & \text{for } n \text{ odd} \\
  n^3 & \text{for } n \text{ even} 
\end{cases} \]

\[ g(n) = n^2 \]

\[ f(n) \not\in O(g(n)) \]
\[ g(n) \not\in O(f(n)) \]
\[ n \in O(f(n)) \]
\[ f(n) \in O(n^3) \]
3-29: **Big-Ω Notation**

\( \Omega(f(n)) \) is the set of all functions that are bound from below by \( f(n) \)

\[ T(n) \in \Omega(f(n)) \text{ if } \exists c, n_0 \text{ such that } T(n) \geq c \cdot f(n) \text{ when } n > n_0 \]
3-30: **Big-Ω Notation**

Ω(f(n)) is the set of all functions that are bound from *below* by f(n)

T(n) ∈ Ω(f(n)) if

\[ \exists c, n_0 \text{ such that } T(n) \geq c \cdot f(n) \text{ when } n > n_0 \]

\[ f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(f(n)) \]
\[ \Theta(f(n)) \] is the set of all functions that are bound \textit{both} above \textit{and} below by \( f(n) \). \( \Theta \) is a \textit{tight bound}

\[ T(n) \in \Omega(f(n)) \] if

\[ T(n) \in O(f(n)) \text{ and } T(n) \in \Omega(f(n)) \]
3-32: Big-Oh Rules

1. If \( f(n) \in O(g(n)) \) and \( g(n) \in O(h(n)) \), then \( f(n) \in O(h(n)) \)

2. If \( f(n) \in O(kg(n)) \) for any constant \( k > 0 \), then \( f(n) \in O(g(n)) \)

3. If \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \), then \( f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n))) \)

4. If \( f_1(n) \in O(g_1(n)) \) and \( f_2(n) \in O(g_2(n)) \), then \( f_1(n) \times f_2(n) \in O(g_1(n) \times g_2(n)) \)

(Also work for \( \Omega \), and hence \( \Theta \))
3-33: **Big-Oh Guidelines**

- Don’t include constants/low order terms in Big-Oh
- Simple statements: $\Theta(1)$
- Loops: $\Theta(\text{inside}) \times \# \text{ of iterations}$
  - Nested loops work the same way
- Consecutive statements: Longest Statement
- Conditional (if) statements:
  $O(\text{Test} + \text{longest branch})$
for (i=1; i<n; i++)
    sum++;
for (i=1; i<n; i++)
    sum++;

Executed n times

O(1)

Running time: $O(n), \Omega(n), \Theta(n)$
for (i=1; i<n; i=i+2)
    sum++;
for (i=1; i<n; i=i+2)  
  sum++;  

Executed n/2 times  
O(1)

Running time: $O(n), \Omega(n), \Theta(n)$
for (i=1; i<n; i++)
    for (j=1; j < n/2; j++)
        sum++;
for (i=1; i<n; i++)
    for (j=1; j < n/2; j++)
        sum++;

Executed n times
Executed n/2 times
O(1)

Running time: \(O(n^2), \Omega(n^2), \Theta(n^2)\)
for (i=1; i<n; i=i*2)
    sum++;
3-41: Calculating Big-Oh

for (i=1; i<n; i=i*2)     Executed \lg n times
    sum++;                 O(1)

Running Time: \( O(\lg n), \Omega(\lg n), \Theta(\lg n) \)
for (i=0; i<n; i++)
    for (j = 0; j<i; j++)
        sum++;
for (i=0; i<n; i++) Executed n times
   for (j = 0; j<i; j++) Executed <= n times
      sum++;

Running Time: $O(n^2)$. Also $\Omega(n^2)$?
for (i=0; i<n; i++)
    for (j = 0; j<i; j++)
        sum++;

Exact # of times sum++ is executed:

\[
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2}
\]

\[\in \Theta(n^2)\]
sum = 0;
for (i=0; i<n; i++)
    sum++;
for (i=1; i<n; i=i*2)
    sum++;
3-46: Calculating Big-Oh

sum = 0;  
for (i=0; i<n; i++) Executed n times  
    sum++; O(1)  
for (i=1; i<n; i=i*2) Executed lg n times  
    sum++; O(1)

Running Time: $O(n), \Omega(n), \Theta(n)$
sum = 0;
for (i=0; i<n; i=i+2)
    sum++;
for (i=0; i<n/2; i=i+5)
    sum++;
sum = 0;
for (i=0; i<n; i=i+2)
    sum++;
for (i=0; i<n/2; i=i+5)
    sum++;

Running Time: $O(n), \Omega(n), \Theta(n)$

3-48: Calculating Big-Oh

O(1)
Executed $n/2$ times
O(1)

O(1)
Executed $n/10$ times
O(1)
for (i=0; i<n; i++)
    for (j=1; j<n; j=j*2)
        for (k=1; k<n; k=k+2)
            sum++;
3-50: Calculating Big-Oh

for (i=0; i<n; i++) Executed n times
   for (j=1; j<n; j=j*2) Executed \( \lg n \) times
      for (k=1; k<n; k=k+2) Executed \( n/2 \) times
         sum++;
         O(1)

Running Time: \( O(n^2 \lg n) \), \( \Omega(n^2 \lg n) \), \( \Theta(n^2 \lg n) \)
sum = 0;
for (i=1; i<n; i=i*2)
  for (j=0; j<n; j++)
    sum++;
Calculating Big-Oh

sum = 0;
for (i=1; i<n; i=i*2)
    for (j=0; j<n; j++)
        sum++;  \( O(1) \)

\[ \text{Execution:} \quad \text{for } (i=1; i<n; i=i \cdot 2) \]
\[ \text{for } (j=0; j<n; j++) \text{, executed } \lg n \text{ times} \]
\[ \text{sum++; } \quad \text{executed } n \text{ times} \]
\[ O(1) \]

Running Time: \( O(n \lg n), \Omega(n \lg n), \Theta(n \lg n) \)
sum = 0;
for (i=1; i<n; i=i*2)
  for (j=0; j<i; j++)
    sum++;
3-54: Calculating Big-Oh

```plaintext
sum = 0;
for (i=1; i<n; i=i*2)
    for (j=0; j<i; j++)
        sum++;
```

$O(1)$

Executed $\log n$ times

Executed $\leq n$ times

$O(1)$

Running Time: $O(n \log n)$. Also $\Omega(n \log n)$?
sum = 0;
for (i=1; i<n; i=i*2)
    for (j=0; j<i; j++)
        sum++;

# of times sum++ is executed:

\[
\sum_{i=0}^{\log_2 n} 2^i = 2^{\log_2 n+1} - 1
\]

\[
= 2n - 1
\]

\[\in \Theta(n)\]

See Chapter 2 for Summations!