Data Structures and Algorithms
CS245-2017S-02
Algorithm Analysis

David Galles

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;
}
```
How long does the following function take to run:

```java
boolean find(int A[], int elem) {
    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?
How long does an algorithm take to run?

• Implement on a computer, time using a stopwatch.
Measuring Time Efficiency

How long does an algorithm take to run?

• Implement on a computer, time using a stopwatch. 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*
How long does an algorithm take to run?

- Implement on a computer, time using a stopwatch. Problems:
  - Not just testing algorithm – testing implementation of algorithm
  - Implementation details (cache performance, other programs running in the background, etc) can affect results
- Better Method: Build a mathematical model of the running time, use model to compare algorithms
02-11: Competing Algorithms

• Linear Search

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

• 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);
}
02-12: Linear vs Binary

• Linear Search

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

Time Required, for a problem of size $n$ (worst case):
02-13: Linear vs Binary

- Linear Search

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

  Time Required, for a problem of size $n$ (worst case):

  $c_1 \cdot n$ for some constant $c_1$
02-14: Linear vs Binary

- Binary Search

```c
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);
}
```

Time Required, for a problem of size $n$ (worst case):
Binary Search

```c
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);
}
```

Time Required, for a problem of size $n$ (worst case): $c_2 \times \lg(n)$ for some constant $c_2$
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 $c_1$?
## 02-17: 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>$10^{10}$</td>
<td>17 minutes</td>
<td>2.3 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 $= $ 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>25</td>
</tr>
</tbody>
</table>
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
  \[ x^3 + x \text{ vs } x^3 \]

We’d like a formal method for describing what is important when analyzing running time, and what is not.
$O(f(n))$ is the set of all functions that are bound from above by $f(n)$

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

\[ \exists c, n_0 \text{ such that } T(n) \leq c \cdot f(n) \text{ when } n > n_0 \]
\( n \in O(n) \) ?
\( 10n \in O(n) \) ?
\( n \in O(10n) \) ?
\( n \in O(n^2) \) ?
\( n^2 \in O(n) \) ?
\( 10n^2 \in O(n^2) \) ?
\( n \log n \in O(n^2) \) ?
\( \log n \in O(n^2) \) ?
\( 3n + 4 \in O(n) \) ?
\( 5n^2 + 10n - 2 \in O(n^3)? O(n^2) ? O(n) ? \)
$n \in O(n)$
$10n \in O(n)$
$n \in O(10n)$
$n \in O(n^2)$
$n^2 \notin O(n)$
$10n^2 \in O(n^2)$
$n \log n \in O(n^2)$
$\log n \in O(n)$
$3n + 4 \in O(n)$
$5n^2 + 10n - 2 \in O(n^3), \in O(n^2), \notin O(n)$
\[
\begin{align*}
\sqrt{n} & \in O(n) \ ? \\
\lg n & \in O(2^n) \ ? \\
\lg n & \in O(n) \ ? \\
n \lg n & \in O(n^2) \ ? \\
\sqrt{n} & \in O(\lg n) \ ? \\
\lg n & \in O(\sqrt{n}) \ ? \\
n \lg n & \in O(n^{3/2}) \ ? \\
n^3 + n \lg n + n \sqrt{n} & \in O(n \lg n) \ ? \\
n^3 + n \lg n + n \sqrt{n} & \in O(n^3) \ ? \\
n^3 + n \lg n + n \sqrt{n} & \in O(n^4) \ ?
\end{align*}
\]
$\sqrt{n} \in O(n)$

$\lg n \in O(2^n)$

$\lg n \in O(n)$

$n \lg n \in O(n^2)$

$\sqrt{n} \notin O(\lg n)$

$\lg n \in O(\sqrt{n})$

$n \lg n \in O(n^{3/2})$

$n^3 + n \lg n + n\sqrt{n} \notin O(n \lg n)$

$n^3 + n \lg n + n\sqrt{n} \in O(n^3)$

$n^3 + n \lg n + n\sqrt{n} \in O(n^4)$
\[ f(n) = \begin{cases} 
  n & \text{for } n \text{ odd} \\
  n^3 & \text{for } n \text{ even} 
\end{cases} \]

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

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

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

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

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

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

\[ f(n) \not\in O(g(n)) \]

\[ g(n) \not\in O(f(n)) \]

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

\[ f(n) \in O(n^3) \]
$\Omega(f(n))$ is the set of all functions that are bound from below by $f(n)$

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

$\exists c, n_0$ such that $T(n) \geq c \times f(n)$ when $n > n_0$
$\Omega(f(n))$ is the set of all functions that are bound from below by $f(n)$

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

$$\exists c, n_0 \text{ such that } T(n) \geq c \times f(n) \text{ when } n > n_0$$

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

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

$T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$
02-31: 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 $f_1(n) \ast f_2(n) \in O(g_1(n) \ast g_2(n))$

(Also work for $\Omega$, and hence $\Theta$)
Don’t include constants/low order terms in Big-Oh

Simple statements: $\Theta(1)$

Loops: $\Theta$(inside) * # of iterations
  - Nested loops work the same way

Consecutive statements: Longest Statement

Conditional (if) statements: $O$(Test + longest branch)
for (i=1; i<n; i++)
    sum++;
Calculating Big-Oh

for (i=1; i<n; i++) Executed n times
    sum++;

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

02-36: Calculating Big-Oh

for (i=1; i<n; i=i+2) sum++;

Executed n/2 times 0(1)

Running time: $O(n), \Omega(n), \Theta(n)$
Calculating Big-Oh

for (i=1; i<n; i++)
    for (j=1; j < n/2; j++)
        sum++;
Calculating Big-Oh

for (i=1; i<n; i++) Executed n times
   for (j=1; j < n/2; j++) Executed n/2 times 0(1)
   sum++;

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

Running Time: $O(\lg n)$, $\Omega(\lg n)$, $\Theta(\lg n)$
Calculating Big-Oh

```plaintext
for (i=0; i<n; i++)
    for (j = 0; j<i; j++)
        sum++;
```
Calculating Big-Oh

for (i=0; i<n; i++) Executed n times
for (j = 0; j<i; j++) Executed <= n times
sum++; 0(1)

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 \texttt{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++;
Calculating Big-Oh

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

Running Time: $O(n)$, $\Omega(n)$, $\Theta(n)$
Calculating Big-Oh

sum = 0;
for (i=0; i<n; i=i+2)
    sum++;
for (i=0; i<n/2; i=i+5)
    sum++;
Calculating Big-Oh

sum = 0;
for (i=0; i<n; i=i+2)
    sum++;
for (i=0; i<n/2; i=i+5)
    sum++;

Running Time: $O(n)$, $\Omega(n)$, $\Theta(n)$
Calculating Big-Oh

for (i=0; i<n; i++)
  for (j=1; j<n; j=j*2)
    for (k=1; k<n; k=k+2)
      sum++;
02-49: Calculating Big-Oh

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

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

Running Time: $O(n^2 \lg n)$, $\Omega(n^2 \lg n)$, $\Theta(n^2 \lg n)$
02-50: Calculating Big-Oh

```
sum = 0;
for (i=1; i<n; i=i*2)
    for (j=0; j<n; j++)
        sum++;
```
Calculating Big-Oh

\[ \text{sum} = 0; \]
\[ \text{for (i=1; i<n; i=i*2) } \quad \text{Executed } \lg n \text{ times} \]
\[ \quad \text{for (j=0; j<n; j++) } \quad \text{Executed } n \text{ times} \]
\[ \text{sum++; } \quad \text{O(1)} \]

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++;

Calculating Big-Oh

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

Running Time: $O(n \log n)$. Also $\Omega(n \log n)$?
Calculating Big-Oh

\[
\text{sum} = 0;
\text{for (i=1; i<n; i=i*2)}
    \text{for (j=0; j<i; j++)}
        \text{sum++;}
\]

\# of times sum++ is executed:

\[
\sum_{i=0}^{\lg n} 2^i = 2^{\lg n+1} - 1 = 2n - 1 \in \Theta(n)
\]
Of course, a little change can mess things up a bit ...

```c
sum = 0;
for (i=1; i<=n; i=i+1)
    for (j=1; j<=i; j=j*2)
        sum++;
```
Of course, a little change can mess things up a bit ...

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

So, this is code is $O(n \lg n)$ – but is it also $\Omega(n \lg n)$?
Of course, a little change can mess things up a bit ...

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

Total time \( \sum_{i=1}^{n} \lg i \) is executed:

This can be tricky to evaluate, but we only need a bound ...
02-58: Calculating Big-Oh

Total # of times \( \text{sum}++ \) is executed:

\[
\sum_{i=1}^{n} \lg i = \sum_{i=1}^{n/2} \lg i + \sum_{i=n/2}^{n} \lg i \\
\geq \sum_{i=n/2}^{n} \lg i \\
= n/2 \lg n/2 \\
\in \Omega(n \lg n)
\]