CS302

**Topic:** Algorithm Analysis #2

*Tuesday, Sept. 27, 2005*
Announcements

- Lab 4 (Stock Reports); due Friday, Oct. 7 (2 weeks)

- Don’t procrastinate!!

  *It’s a job that’s never started*
  
  *that takes the longest to finish.*

  -- J.R.R. Tolkien

- No office hours today (I have a Dean’s advisory council meeting to attend) – send email if you need to see me, and we’ll set up something for tomorrow
Recall: mathematical definition of “Big-O”

\[ O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0, \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \} \]
Comparing running times

- Put these in order of increasing growth rate:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Growth Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lg^2 n$</td>
<td>$1$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$\lg^2 n$</td>
</tr>
<tr>
<td>$1$</td>
<td>$n \lg n$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>$n \lg n$</td>
<td>$n^3$</td>
</tr>
<tr>
<td>$2^{2^n}$</td>
<td>$2^n$</td>
</tr>
<tr>
<td>$\lg n$</td>
<td>$n!$</td>
</tr>
<tr>
<td>$n!$</td>
<td>$2^{2^n}$</td>
</tr>
</tbody>
</table>
How do the following compare?

\[ n \log n \quad \text{vs.} \quad n \log (n^2) \]

They grow at same rate:

\[ n \log (n^2) = 2n \log n \rightarrow O(n \log n) \]
An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 if the running time is the following:

- **linear**: 5 times as long (i.e., 2.5 ms)
- **$O(n \log n)$**: slightly more than 5 times as long
- **quadratic**: 25 times as long (i.e., 12.5 ms)
- **cubic**: 125 times as long (i.e., 62.5 ms)
We typically want tight bounds

Example:

```cpp
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        k++;
```

Runtime = $O(n^2)$

So, does this mean that “Runtime = $O(n^3)$” is false?

- No! $O(n^3)$ is true, but isn’t as tight of a bound as we can define
- We prefer $O(n^2)$, because it is a tighter bound
Other asymptotic notation: \( o \) (“little oh”) (upper bound that isn’t asymptotically tight)

\[
o(g(n)) = \{ f(n): \text{for any positive constant } c, \\
\text{there exists a positive constant } n_0, \text{ such that } \\
0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}
\]

Compare to “big-O” definition:

\[
O(g(n)) = \{ f(n): \text{there exist positive constants } c \\
\text{and } n_0, \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}
\]

- Example:
  
  ```
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      k++;
  ```

- Runtime = \( O(n^2) \)
- Runtime = \( o(n^3) \)

Intuitively, in \( o \)-notation, \( f(n) \) becomes insignificant relative to \( g(n) \) as \( n \) approaches infinity:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]
Other asymptotic notation: 
\( \Omega \) Lower bound

\[ \Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0, \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} \]

\( \omega \) is “little omega”, i.e., lower bound that isn’t asymptotically tight

(this is analog to o-notation)
Comparing relative growth rates

- We can always determine relative growth rates of two functions by computing:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)}
\]

[use L’Hôpital’s rule, if necessary:

If \( \lim_{n \to \infty} f(n) = \infty \), and \( \lim_{n \to \infty} g(n) = \infty \), then:

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}
\]

where \( f'(n) \) and \( g'(n) \) are the derivatives of \( f(n) \) and \( g(n) \).]
Comparing relative growth rates

- We can always determine relative growth rates of two functions by computing:

  \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} \]

- Limit = 0 \(\Rightarrow f(n) = o(g(n)) \) or \( g(n) = \omega(f(n)) \)

- Limit = \( c \neq 0 \) \(\Rightarrow f(n) = \Theta(g(n)) \)

- Limit = \( \infty \) \(\Rightarrow g(n) = o(f(n)) \) or \( f(n) = \omega(g(n)) \)

- Limit oscillates: No relation
Comparing running times

- Suppose $T_1(n) = O(f(n))$ and $T_2(n) = O(f(n))$

Which of the following are true:

1. $T_1(n) + T_2(n) = O(f(n))$ **TRUE**

2. $T_1(n) - T_2(n) = o(f(n))$ **FALSE** Counterexample: $T_1(n) = 2n; T_1(n) = n; f(n) = n$

3. $T_1(n) / T_2(n) = O(1)$ **FALSE** Counterexample: $T_1(n) = n^2; T_1(n) = n; f(n) = n^2$

4. $T_1(n) = O(T_2(n))$ **FALSE** Counterexample: (same as #3 above)
General rules for determining computational complexity of an alg.

- **For loops:**
  - Running time is at most the running time of the statements inside the for loop (including tests) times the number of iterations

- **Nested loops:**
  - Analyze from the inside out. Total running time is the running time of the interior statements times the product of the sizes of the loops
  - Example:
    ```java
    for (i=0; i<n; i++)
      for (j=0; j<n; j++)
        k++;
    
    Runtime?
    O(n^2)
    ```
Consecutive statements:

- Just add them up, keeping the maximum value
- Example:

```c
for (i=0; i<n; i++)
a[i] = 0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i] += a[j] + i + j;
```

Runtime?

$O(n^2)$
General rules, con’t.

- If-Else statements:
  - Running time is at most the running time of the test plus the larger of the running times of the two parts of the conditional
  - Example:
    ```c
    if (test == true)
        a[0] = 0;
    else {
        for (i=0; i<n; i++)
            a[i] += a[i+1];
    }
    
    Runtime?
    
    O(n)
    ```
General rules, con’t.

- Function calls
  - Analyze first, using previous rules

- Recursive function calls
  - Check to see if it’s equivalent to a “for” loop, and if so, analyze accordingly
  - Otherwise, set up recurrence and solve (e.g., using recursion tree)
Example 1: What is the runtime?

\[
\begin{align*}
\text{O}(1) & \quad \text{sum} = 0; \\
\text{O}(n) & \quad \text{for} \ (i = 0; i < n; i++) \\
\text{O}(1) & \quad \text{sum}++; \\
\end{align*}
\]

\[
T(n) = \text{O}(1) + \text{O}(n \times 1)
\]

\[
T(n) = \text{O}(n)
\]
Example 2: What is the runtime?

$\text{O}(1) \quad \text{sum} = 0;$

$\text{O}(n) \quad \text{for } (i = 0; i < n; i++)$

$\text{O}(n) \quad \text{for } (j=0; j < n; j++)$

$\text{O}(1) \quad \text{sum++;}$

$T(n) = \text{O}(1) + \text{O}(n \times n \times 1)$

$T(n) = \text{O}(n^2)$
Example 3: What is the runtime?

\[
O(1) \quad \text{sum} = 0;
\]

\[
O(n) \quad \text{for} \ (i = 0; \ i < n; \ i++)
\]

\[
O(n^2) \quad \text{for} \ (j=0; \ j < n \times n; \ j++)
\]

\[
O(1) \quad \text{sum}++; 
\]

\[
T(n) = O(1) + O(n \times n^2 \times 1)
\]

\[
T(n) = O(n^3)
\]
Example 4: What is the runtime?

O(1) \( \text{sum} = 0; \)

for (i = 0; i < n; i++)
   \( \text{for (j=0; j < i; j++)} \)
   O(1) sum++;

\[ \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n} i = \frac{1}{2} n(n+1) = O(n^2) \]

\( \Rightarrow T(n) = O(1) + O(n^2 \times 1) = O(n^2) \)

Easier way: we could also approximate runtime by looking at max values for each looping variable
\( \Rightarrow O(1) + O(n \times n \times 1) = O(n^2) \)
Example 5: What is the runtime?

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

\[ T(n) = O(1) + O(n \times n^2 \times n^2 \times 1) \]

\[ T(n) = O(n^5) \]
Example 6: What is the runtime?

MyFunc(int m, n) // Assume m and n are approx. equal
    if ((m = 0) or (n = 0))
        then return m
    else
        return min{MyFunc(m-1,n),
                     MyFunc(m-1, n-1),
                     MyFunc(m, n-1)} + 1
Example 6: What is the runtime?

MyFunc(int m, n) // Assume m and n are approx. equal
if ((m = 0) or (n = 0))
  then return m
else
  return \min\{MyFunc(m-1,n), MyFunc(m-1, n-1), MyFunc(m, n-1)\} + 1
T(m,n)
Example 6: What is the runtime?

MyFunc(int \( m, n \)) // Assume \( m \) and \( n \) are approx. equal
if ((\( m = 0 \)) or (\( n = 0 \))
then return \( m \)
else
return \( \min\{\text{MyFunc}(m-1,n), \text{MyFunc}(m-1, n-1), \text{MyFunc}(m, n-1)\}\) + 1

\[
\begin{align*}
&\text{C} \\
&\text{T}(m-1,n) \quad \text{T}(m-1,n-1) \quad \text{T}(m,n-1)
\end{align*}
\]
Example 6: What is the runtime?

MyFunc(int m, n) // Assume m and n are approx.
equal
if ((m = 0) or (n = 0))
    then return m
else
    return min{MyFunc(m-1,n),
                MyFunc(m-1, n-1),
                MyFunc(m, n-1)} + 1
Example 6: What is the runtime?

MyFunc(int \( m, n \)) \hspace{1cm} // Assume \( m \) and \( n \) are approx. equal

if \( ((m = 0) \text{ or } (n = 0)) \)
then return \( m \)
else
    return \( \min\{\text{MyFunc}(m-1,n), \text{MyFunc}(m-1, n-1), \text{MyFunc}(m, n-1)\} + 1 \)

\( h = \mathcal{O}(n) \text{ or } \mathcal{O}(m) \)

\( T(m-2,n) \quad T(m-2,n-1) \quad T(m-2,n-2) \quad T(m-1,n-2) \quad =\mathcal{O}(3^n) = \mathcal{O}(2^n) \)
Efficient Exponentiation

- Obvious algorithm for computing $x^n$ uses $n-1$ multiplications
- Better way:
  - Observation:
    - N even $\Rightarrow x^n = x^{n/2} \times x^{n/2}$
    - N odd $\Rightarrow x^n = x^{(n-1)/2} \times x^{(n-1)/2} \times x$

```c
long pow( long x, int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return x;
    if (isEven(n))
        return pow(x*x, n/2);
    else
        return pow(x*x, n/2) * x;
}
```

What calculations are done for $x^{62}$?

- $x^{62} = (x^{31})^2$
- $x^{31} = (x^{15})^2$
- $x^{15} = (x^7)^2$
- $x^7 = (x^3)^2$
- $x^3 = (x^2)x$

= 9 multiplications
Considerations in choosing among competing algorithms

1. Which is fastest?
   - Asymptotic analysis only provides a rough guide. Often, you have to implement the algorithms to determine which is faster.

2. Which is simplest to implement?
   - Often, if 2 algorithms have comparable running times, but one is far easier to implement and maintain, the simpler algorithm will be chosen.

3. How often will the algorithm be executed?
   - Programmer time may be more important than computer time.

4. What is the size of the input?
   - If input size is small, algorithm with a higher time complexity but smaller constants may be preferrable.
That’s all, folks.

☐ Questions?