CS 281
Prof. Alex Pothen
Webbook:

http://www.cs.odu.edu/~webbook
http://www.cs.odu.edu/~pothen/Courses/CS281
Mon Sep 21, 1998
Lecture 6
Running Times of Algorithms
Sec. 2.2 and 1.8
Next Lecture:
Sec. 1.8, Appendix 1
Linear Search algorithm

LinearSearch := proc(A::list(integer),
    item::integer)
  i := 1;
  while (i ≤ n and A[i] ≠ item) do
    i := i + 1;
  od
  if i < n + 1 then
    item is in position i;
  else
    item is not found;
  fi
**Sentinel Search algorithm**

SentinelSearch := proc(A::list(integer),
                      item::integer)
    \[ A[n + 1] := \text{item}; \]
    \[ i := 1; \]
    \[ \text{while } (A[i] \neq \text{item}) \text{ do} \]
    \[ \quad i := i + 1; \]
    \[ \text{od} \]
    \[ \text{if } i < n + 1 \text{ then} \]
    \[ \quad \text{item is in position } i; \]
    \[ \text{else} \]
    \[ \quad \text{item is not found}; \]
    \[ \text{fi} \]
Running Times

The running time of a program depends on computer operating system, compiler input other programs running on computer Thus difficult to predict exact time required by a program.

Predict running times to within a constant factor that accounts for the computer, OS, and compiler.

Worst case running times: Predict the maximum time required over all inputs.
Big Oh

Measure worst-case running times as a function of input size, $n$:

Count assignments or comparisons in while loop
linear search: $3n$
sentinel search: $2(n + 1)$

We say: $3n$ is $O(n)$. (Big Oh of $n$).
$2(n + 1) = 2n + 2 = O(n)$.
$6\lceil\log_2 n\rceil \leq 6(\log_2 n + 1) = 6\log_2 n + 6$
is $O(\log_2 n)$ (Big Oh of $\log_2 n$).
Big Oh and Worst-case Running Times

Let $T(n)$ be a function measuring the worst-case running time of a program, as a function of input size $n$. Then

$n$ is a non-negative integer
$T(n)$ is a non-negative function.

The Big Oh concept measures the growth rate of the running time $T(n)$ of an algorithm, for large values of $n$.

Hence we ignore the behavior of $T(n)$ for small values of $n$: $n < n_0$. Consider only the term determining the growth rate.

A figure to illustrate the Big-Oh concept.
Examples

- $n + 2 \leq n + n = 2n$, for all $n \geq 2$.
  Hence $n + 2 = O(n)$: $n_0 = 1$, $c = 2$.

- $n + 2 \leq n + (n/2) = (3n/2)$, for all $n \geq 4$.
  Hence $n + 2 = O(n)$: $n_0 = 4$, $c = 3/2$.

- Note from these two examples that $n_0$ and $c$ can be chosen in different ways!
More Examples

• $\frac{1}{2}(n^2 + n) \leq \frac{1}{2}(n^2 + n^2) = n^2$, for all $n \geq 1$.
  Hence $n^2 + n = O(n^2)$: $n_0 = 1$, $c = 1$.

• $2(\log_2 n + 2) \leq 2(\log_2 n + \log_2 n) = 4 \log_2 n$, for all $n \geq 4$.
  Hence $2(\log_2 n + 2) = O(\log_2 n)$: $n_0 = 4$, $c = 4$.

• Is $(n + 1)^2 = O(n^2)$? Is it $O(n^3)$?

• Is $(2n + 3)^2 = O((n + 1)^2)$?
Template for a Big Oh Proof

- Find the values of the witnesses:
  nonnegative integer $n_0$
  constant $c$.
  Do this by looking at the expression for $T(n)$.

- Show by algebraic manipulation that if $n \geq n_0$, then $T(n) \leq cf(n)$, for the particular witnesses chosen.

We try to find the “tightest” Big Oh bound. This means we choose $f(n)$ to be the smallest function that is an upper bound on $T(n)$, while ignoring constants.
## Time Complexity

<table>
<thead>
<tr>
<th>Big Oh</th>
<th>Informal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
</tr>
<tr>
<td>$O(\log_2 n)$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
</tr>
<tr>
<td>$O(n \log_2 n)$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>cubic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>exponential</td>
</tr>
<tr>
<td>$O(B^n)$</td>
<td>$(B &gt; 1)$ exponential</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>$n$-factorial</td>
</tr>
</tbody>
</table>

Exponential algorithms are impractical for large problem sizes; (low-degree) polynomial algorithms are practical.