Introduction to Computers & Programming

Asymptotic analysis: upper/ lower bounds, $\Theta$ notation
Binary, Insertion, and Merge sort
Complexity Analysis

- Complexity: rate at which storage or time grows as a function of the problem size
  - Growth depends on compiler, machine, …

- Asymptotic analysis
  - Describing the inherent complexity of a program, independent of machine and compiler.
  - A “proportionality” approach, expressing the complexity in terms of its relationship to some known function.
Asymptotic Analysis

- **Idea**: as problem size grows, the complexity can be described as a simple *proportionality* to some known function.

“Big Oh”
Asymptotic Analysis: Big-oh

Definition: \( T(n) = O(f(n)) \) -- \( T \) of \( n \) is in Big Oh of \( f \) of \( n \)
iff there are constants \( c \) and \( n_0 \) such that \( T(n) \leq cf(n) \) for all \( n \geq n_0 \)

Usage: The algorithm is in \( O(n^2) \) in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., \( n>n_0 \)), the algorithm always executes in less than \( cf(n) \) steps in [best, average, worst] case.

Big oh is said to describe an “upper bound” on the complexity.
Big-oh Notation (cont)

Big-oh notation indicates an **upper bound** on the complexity.

**Example**: If $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.

**Wish tightest upper bound**: While $T(n) = 3n^2$ is in $O(n^{25})$, we prefer $O(n^2)$. 
- \( T(n) = O(1) \) \textit{constant} growth
  - E.g., array access
- \( T(n) = O(\log(n)) \) \textit{logarithmic} growth
  - E.g., binary search
- \( T(n) = O(n) \) \textit{linear} growth
  - E.g., looping over all elements in a 1-dimensional array
- \( T(n) = O(n \log n) \) “\( n \log n \)” growth
  - E.g., Merge sort
- \( T(n) = O(n^k) \) \textit{polynomial} growth
  - E.g., Selection sort is \( n^2 \), \( k \) seldom larger than 5
- \( T(n) = O(2^n) \) \textit{exponential} growth
  - E.g., basically useless for anything but very small problems
1 billion instructions/second

<table>
<thead>
<tr>
<th>n</th>
<th>T(n)= n</th>
<th>T(n)= n \log(n)</th>
<th>T(n)= n^2</th>
<th>T(n)= n^3</th>
<th>T(n)= 2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.005 µs</td>
<td>0.01 µs</td>
<td>0.03 µs</td>
<td>0.13 µs</td>
<td>0.03 µs</td>
</tr>
<tr>
<td>10</td>
<td>0.1 µs</td>
<td>0.03 µs</td>
<td>0.1 µs</td>
<td>1 µs</td>
<td>1 µs</td>
</tr>
<tr>
<td>20</td>
<td>0.02 µs</td>
<td>0.09 µs</td>
<td>0.4 µs</td>
<td>8 µs</td>
<td>1ms</td>
</tr>
<tr>
<td>50</td>
<td>0.05 µs</td>
<td>0.28 µs</td>
<td>2.5 µs</td>
<td>125 µs</td>
<td>13 days</td>
</tr>
<tr>
<td>100</td>
<td>0.1 µs</td>
<td>0.66 µs</td>
<td>10 µs</td>
<td>1 ms</td>
<td>4 \times 10^{13} years</td>
</tr>
</tbody>
</table>
Big-Oh Examples

Example 1: Finding value $X$ in an array (average cost).

$T(n) = c_sn/2$.

For all values of $n > 1$, $c_sn/2 \leq c_sn$.

Therefore, by the definition, $T(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$. 
Big-Oh Examples

Example 2: \( T(n) = c_1 n^2 + c_2 n \) in average case.

\[
c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2 \text{ for all } n > 1.
\]

\( T(n) \leq cn^2 \text{ for } c = c_1 + c_2 \) and \( n_0 = 1 \).

Therefore, \( T(n) \) is in \( O(n^2) \) by the definition.

Example 3: \( T(n) = c \). We say this is in \( O(1) \).
Does Big-oh tell the whole story?

- $T_1(n) = T_2(n) = O(g(n))$
- $T_1(n) = 50 + 3n + (10 + 5 + 15)n = 50 + 33n$
  - Setup of algorithm -- takes 50 time units
  - read n elements into array -- 3 units/element
  - for i 1..n
    - do operation1 on $A[i]$ -- takes 10 units
    - do operation2 on $A[i]$ -- takes 5 units
    - do operation3 on $A[i]$ -- takes 15 units
- $T_2(n) = 200 + 3n + (10 + 5)n = 200 + 18n$
  - Setup of algorithm -- takes 200 time units
  - read n elements into array -- 3 units/element
  - for i 1..n
    - do operation1 on $A[i]$ -- takes 10 units
    - do operation2 on $A[i]$ -- takes 5 units
**Big-Omega**

**Definition:** For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \geq cg(n)$ for all $n > n_0$.

**Meaning:** For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $cg(n)$ steps.

**Lower bound.**
Big-Omega Example

\[ T(n) = c_1 n^2 + c_2 n. \]

\[ c_1 n^2 + c_2 n \geq c_1 n^2 \text{ for all } n > 1. \]
\[ T(n) \geq cn^2 \text{ for } c = c_1 \text{ and } n_0 = 1. \]

Therefore, \( T(n) \) is in \( \Omega(n^2) \) by the definition.

We want the greatest lower bound.
Theta Notation

When big-Oh and $\Omega$ meet, we indicate this by using $\Theta$ (big-Theta) notation.

**Definition:** An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

**Tight bound.**
A Common Misunderstanding

Confusing worst case with upper bound.

**Upper bound** refers to a growth rate.

**Worst case** refers to the worst input from among the choices for possible inputs of a given size.
Simplifying Rules

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

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

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

4. If \( f_1(n) \) is in \( O(g_1(n)) \) and \( f_2(n) \) is in \( O(g_2(n)) \) then \( f_1(n)f_2(n) \) is in \( O(g_1(n)g_2(n)) \).
Binary Search

<table>
<thead>
<tr>
<th>Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>11</td>
<td>13</td>
<td>21</td>
<td>26</td>
<td>29</td>
<td>36</td>
<td>40</td>
<td>41</td>
<td>45</td>
<td>51</td>
<td>54</td>
<td>56</td>
<td>65</td>
<td>72</td>
<td>77</td>
<td>83</td>
</tr>
</tbody>
</table>

How many elements are examined in worst case?
Procedure Binary_Search (Input_Array, Number_To_Search, Return_Index);

Begin
    Set Return_Index to -1;
    Set Current_Index to (Upper_Bound - Lower_Bound + 1) /2.

    Loop
        if the lower_bound > upper_bound
            Exit;
        end if
        if ( Input_Array(Current_Index) = Number_to_search)
            then
                Return_Index := Current_Index
                Exit;
            end if
        if ( Input_Array(Current_Index) < Number_to_search)
            then
                Lower_Bound := Current_Index +1
            else
                Upper_Bound := Current_Index - 1
            end if
        end loop
    end Binary_Search;
Insertion sort

- InsertionSort(A, n) -- A[1..n]
  for j 2..n
    do key := A[j]
      i := j-1
      while i > 0 and A[i] > key
        A[i+1] := A[i]
        i := i-1
      A[i+1] := key
Insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6
2 3 4 6 8 9
Insertion sort

- Running time
  - Depends on the input
    - An already sorted sequence is easier to sort
  - Worst case: input reverse sorted
    \[ T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \]
  - Average case: all permutations equally likely
    \[ T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2) \]
Merge sort

- MergeSort A[1..n]
  1. If the input sequence has only one element, return

  2. Partition the input sequence into two halves

  3. Sort the two subsequences using the same algorithm

  4. Merge the two sorted subsequences to form the output sequence
Merge sort

1 2
20 12
13 11
7 9
2

1 2
20 12
13 11
7 9
1

7

1 2
20 12
13 11
7 9

Merge sort

1 2 7 9 11 12
Merge sort

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases} \]

... 

Total = \Theta(n \ lg \ n)
Space Bounds

- Space bounds can also be analyzed with asymptotic complexity analysis.

**Time**: Algorithm

**Space**: Data Structure
Space/Time Tradeoff Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa.
  - Encoding or packing information
    - Boolean flags
  - Table lookup
    - Factorial
- Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.