CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Algorithmic Complexity I

Department of Computer Science
University of Maryland, College Park
Algorithm Efficiency

- Efficiency
  - Amount of resources used by algorithm
    - Time, space
  - Measuring efficiency
    - Benchmarking
      - Approach
        - Pick some desired inputs
        - Actually run implementation of algorithm
        - Measure time & space needed
    - Asymptotic analysis
Benchmarking

- Advantages
  - Precise information for given configuration
    - Implementation, hardware, inputs

- Disadvantages
  - Affected by configuration
    - Data sets (often too small)
      - Dataset that was the right size 3 years ago is likely too small now
  - Hardware
  - Software
  - Affected by special cases (biased inputs)
  - Does not measure intrinsic efficiency
Asymptotic Analysis

• Approach
  • Mathematically analyze efficiency
  • Calculate time as function of input size \( n \)
    • \( T \approx O( f(n) ) \)
    • \( T \) is on the order of \( f(n) \)
    • “Big O” notation

• Advantages
  • Measures intrinsic efficiency
  • **Dominates efficiency for large input sizes**
  • Programming language, compiler, processor irrelevant
Search Comparison

- For number between 1…100
  - Simple algorithm = 50 steps
  - Binary search algorithm = \( \log_2(n) = 7 \) steps
- For number between 1…100,000
  - Simple algorithm = 50,000 steps
  - Binary search algorithm = \( \log_2(n) \) (about 17 steps)
- Binary search is much more efficient!
Asymptotic Complexity

- Comparing two linear functions

<table>
<thead>
<tr>
<th>Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n/2</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
</tr>
</tbody>
</table>
Asymptotic Complexity

• Comparing two functions
  • $n/2$ and $4n+3$ behave similarly
  • Run time roughly doubles as input size doubles
  • Run time increases linearly with input size
• For large values of $n$
  • $\frac{\text{Time}(2n)}{\text{Time}(n)}$ approaches exactly 2
• Both are $O(n)$ programs
• Example: $2n + 100 \rightarrow O(n)$ (next slide)
Complexity Example

- $2n + 100 \Rightarrow O(n)$
Asymptotic Complexity

- Comparing two quadratic functions

<table>
<thead>
<tr>
<th>Size</th>
<th>n²</th>
<th>2 n² + 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>132</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>520</td>
</tr>
</tbody>
</table>
Asymptotic Complexity

• Comparing two functions
  • \( n^2 \) and \( 2n^2 + 8 \) behave similarly
  • Run time roughly increases by 4 as input size doubles
  • Run time increases quadratically with input size
• For large values of \( n \)
  • \( \text{Time}(2n) / \text{Time}(n) \) approaches 4
• Both are \( O( n^2 ) \) programs
• Example: \( \frac{1}{2} n^2 + 100n \rightarrow O(n^2) \) (next slide)
Complexity Examples

- $\frac{1}{2} n^2 + 100 n \Rightarrow O(n^2)$
Asymptotic Complexity

- Comparing two log functions

<table>
<thead>
<tr>
<th>Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>log&lt;sub&gt;2&lt;/sub&gt;( n )</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
</tr>
</tbody>
</table>
Asymptotic Complexity

- Comparing two functions
  - $\log_2(n)$ and $5 \times \log_2(n) + 3$ behave similarly
  - Run time roughly increases by constant as input size doubles
  - Run time increases logarithmically with input size
- For large values of $n$
  - $\text{Time}(2n) - \text{Time}(n)$ approaches constant
  - Base of logarithm does not matter
    - Simply a multiplicative factor
      $\log_a N = (\log_b N) / (\log_b a)$
  - Both are $O(\log(n))$ programs
Big-O Notation

- Represents
  - Upper bound on number of steps in algorithm
    - For sufficiently large input size
  - Intrinsic efficiency of algorithm for large inputs
Formal Definition of Big-O

- Function \( f(n) \) is \( O\left( g(n) \right) \) if
  - For some positive constants \( M, N_0 \)
  - \( M \times g(n) \geq f(n) \), for all \( n \geq N_0 \)
- Intuitively
  - For some coefficient \( M \) & all data sizes \( \geq N_0 \)
    - \( M \times g(n) \) is always greater than \( f(n) \)
Big-O Examples

- $2n^2 + 10n + 1000 \Rightarrow O(n^2)$
  - Select $M = 4$, $N_0 = 100$
  - For $n \geq 100$
    - $4n^2 \geq 2n^2 + 10n + 1000$ is always true
  - Example $\Rightarrow$ for $n = 100$
    - $40000 \geq 20000 + 1000 + 1000$
Observations

- For large values of n
  - Any $O(\log(n))$ algorithm is faster than $O(n)$
  - Any $O(n)$ algorithm is faster than $O(n^2)$
- Asymptotic complexity is fundamental measure of efficiency
- Big-O results only valid for big values of n
# Asymptotic Complexity Categories

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
<td>Array access</td>
</tr>
<tr>
<td>$O(\log(n))$</td>
<td>Logarithmic</td>
<td>Binary search</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
<td>Largest element</td>
</tr>
<tr>
<td>$O(n \log(n))$</td>
<td>N log N</td>
<td>Optimal sort</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
<td>2D Matrix addition</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
<td>2D Matrix multiply</td>
</tr>
<tr>
<td>$O(n^k)$</td>
<td>Polynomial</td>
<td>Linear programming</td>
</tr>
<tr>
<td>$O(k^n)$</td>
<td>Exponential</td>
<td>Integer programming</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>Factorial</td>
<td>Brute-force search TSP</td>
</tr>
<tr>
<td>$O(n^n)$</td>
<td>N to the N</td>
<td></td>
</tr>
</tbody>
</table>

From smallest to largest, for size $n$, constant $k > 1$
Complexity Category Example

Problem Size vs. # of Solution Steps

- $2^n$
- $n^2$
- $n\log(n)$
- $n$
- $\log(n)$
Complexity Category Example

Problem Size

- \(2^n\)
- \(n^2\)
- \(n\log(n)\)
- \(n\)
- \(\log(n)\)
Calculating Asymptotic Complexity

- As \( n \) increases
  - Highest complexity term dominates
  - Can ignore lower complexity terms
- Examples
  - \( 2n + 100 \) \( \Rightarrow O(n) \)
  - \( 10n + n\log(n) \) \( \Rightarrow O(n\log(n)) \)
  - \( 100n + \frac{1}{2}n^2 \) \( \Rightarrow O(n^2) \)
  - \( 100n^2 + n^3 \) \( \Rightarrow O(n^3) \)
  - \( \frac{1}{100}2^n + 100n^4 \) \( \Rightarrow O(2^n) \)
Types of Case Analysis

- Can analyze different types (cases) of algorithm behavior
- Types of analysis
  - Best case
  - Worst case
  - Average case
  - Amortized
Best/Worst Case Analysis

• Best case
  • Smallest number of steps required
  • Not very useful
  • Example ⇒ Find item in first place checked

• Worst case
  • Largest number of steps required
  • Useful for upper bound on worst performance
    • Real-time applications (e.g., multimedia)
    • Quality of service guarantee
  • Example ⇒ Find item in last place checked
Quicksort Example

- Quicksort
  - One of the fastest comparison sorts
  - Frequently used in practice

- Quicksort algorithm
  - Pick \textit{pivot} value from list
  - Partition list into values smaller & bigger than pivot
  - Recursively sort both lists

- Quicksort properties
  - Average case $= O(n \log(n))$
  - Worst case $= O(n^2)$
    - Pivot $\approx$ smallest / largest value in list
    - Picking from front of nearly sorted list

- Can avoid worst-case behavior
  - Select random pivot value
Average Case Analysis

- **Average case analysis**
  - Number of steps required for “typical” case
  - Most useful metric in practice
  - Different approaches: average case, expected case

- **Average case**
  - Average over all possible inputs
    - Assumes all inputs have the same probability
  - Example
    - Case 1 = 10 steps, Case 2 = 20 steps
    - Average = 15 steps

- **Expected case**
  - Weighted average over all possible inputs
    - Based on probability of each input
  - Example
    - Case 1 (90%) = 10 steps, Case 2 (10%) = 20 steps
    - Average = 11 steps
Amortized Analysis

• Approach
  • Applies to worst-case sequences of operations
  • Finds average running time per operation
  • Example
    • Normal case = 10 steps
    • Every 10\textsuperscript{th} case may require 20 steps
    • Amortized time = 11 steps

• Assumptions
  • Can predict possible sequence of operations
  • Know when worst-case operations are needed
    • Does not require knowledge of probability
  • By using amortized analysis we can show the best way to grow an array is by doubling its size (rather than increasing by adding one entry at a time)
Complexity Category Example

- $2^n$
- $n^2$
- $n \log(n)$
- $n$
- $\log(n)$
Complexity Category Example

The graph shows the relationship between problem size and the number of solution steps for different complexity categories:

- \(2^n\)
- \(n^2\)
- \(n\log(n)\)
- \(n\)
- \(\log(n)\)

The x-axis represents the problem size, and the y-axis represents the number of solution steps. The lines and markers correspond to the different complexity categories as indicated by the legend.