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
  - Asymptotic analysis
Benchmarking

Approach
- Pick some desired inputs
- Actually run implementation of algorithm
- Measure time & space needed

Industry benchmarks
- SPEC – CPU performance
- MySQL – Database applications
- WinStone – Windows PC applications
- MediaBench – Multimedia applications
- Linpack – Numerical scientific applications
Benchmarking

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

Disadvantages
- Affected by configuration
  - Data sets (often too small)
  - A 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 Example

Number guessing game

- Pick a number between 1…n
- Guess a number
- Answer “correct”, “too high”, “too low”
- Repeat guesses until correct number guessed
Linear Search Algorithm

**Algorithm**
- Guess number = 1
- If incorrect, increment guess by 1
- Repeat until correct

**Example**
- Given number between 1…100
- Pick 20
- Guess sequence = 1, 2, 3, 4 … 20
- Required 20 guesses
Linear Search Algorithm

Analysis of # of guesses needed for 1…n

- If number = 1, requires 1 guess
- If number = n, requires n guesses
- On average, needs n/2 guesses
- Time = $O(n) = \text{Linear time}$
Binary Search Algorithm

Algorithm

- Set low and high to be lowest and highest possible value
- Guess middle = (low+high)/2
- If too large, set high = middle-1
- If too small, set low = middle+1
- Repeat until guess correct
Binary Search Algorithm

Example

- Given number between 1…100
- Secret number we are trying to find is 20

Guesses

- low = 1, high = 100, guess 50, Answer = too large
- low = 1, high = 49, guess 25, Answer = too large
- low = 1, high = 24, guess 12, Answer = too small
- low = 13, high = 24, guess 18, Answer = too small
- low = 19, high = 24, guess 21, Answer = too large
- low = 19, high = 20, guess 19, Answer = too small
- low = 20, high = 20, guess 20, Answer = correct

Required 7 guesses
Binary Search Algorithm

Analysis of # of guesses needed for 1…n

- If number = n/2, requires 1 guess
- If number = 1, requires $\log_2(n)$ guesses
- If number = n, requires $\log_2(n)$ guesses
- On average, needs $\log_2(n)$ guesses
- Time = $O(\log_2(n)) = O(\log(n)) = \text{Log time}$
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>n/2</td>
<td>4n+3</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
Asymptotic Complexity

Comparing two log functions

<table>
<thead>
<tr>
<th>Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\log_2(n)$</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 \( 5 \cdot \log_2(n) + 3 \) 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 \( \mathcal{O}(\log(n)) \) programs
Asymptotic Complexity

Comparing two quadratic functions

<table>
<thead>
<tr>
<th>Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n^2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>256</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 $n$
  - $\frac{\text{Time}(2n)}{\text{Time}(n)}$ approaches 4

- Both are $O(n^2)$ 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

\[ f(n) \leq O(\ldots) \]

# steps vs. input size
Formal Definition of Big-O

Function \( f(n) \) is \( O( g(n) ) \) 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**

5n + 1000 ⇒ O(n)

- Select M = 6, N₀ = 1000
- For n ≥ 1000
  - 6n ≥ 5n + 1000 is always true
- Example ⇒ for n = 1000
  - 6000 ≥ 5000 + 1000
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

- Big O categories
  - $O(\log(n))$
  - $O(n)$
  - $O(n^2)$

- 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
Comparison of Complexity

A Comparison of Orders

- $n$
- $\frac{1}{2}n^2$
- $n^3$

$f(x)$ vs. $n$
Complexity Category Example

<table>
<thead>
<tr>
<th>Problem Size</th>
<th># of Solution Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$2^2 = 4$</td>
</tr>
<tr>
<td>3</td>
<td>$3^2 = 9$</td>
</tr>
<tr>
<td>4</td>
<td>$4^2 = 16$</td>
</tr>
<tr>
<td>5</td>
<td>$5^2 = 25$</td>
</tr>
<tr>
<td>6</td>
<td>$6^2 = 36$</td>
</tr>
<tr>
<td>7</td>
<td>$7^2 = 49$</td>
</tr>
<tr>
<td>8</td>
<td>$8^2 = 64$</td>
</tr>
</tbody>
</table>

- $2^n$ growth:
  - Problem Size: 2, 3, 4, 5, 6, 7, 8
  - Solution Steps: 4, 9, 16, 25, 36, 49, 64

- $n^2$ growth:
  - Problem Size: 2, 3, 4, 5, 6, 7, 8
  - Solution Steps: 4, 9, 16, 25, 36, 49, 64

- $n\log(n)$ growth:
  - Problem Size: 2, 3, 4, 5, 6, 7, 8
  - Solution Steps: 2, 3, 4, 5, 6, 7, 8

- $n$ growth:
  - Problem Size: 2, 3, 4, 5, 6, 7, 8
  - Solution Steps: 2, 3, 4, 5, 6, 7, 8

- $\log(n)$ growth:
  - Problem Size: 2, 3, 4, 5, 6, 7, 8
  - Solution Steps: 0, 1, 1, 1, 1, 1, 1
## Complexity Category Example

<table>
<thead>
<tr>
<th>Problem Size</th>
<th># of Solution Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$2^n$</td>
</tr>
<tr>
<td>3</td>
<td>$n^2$</td>
</tr>
<tr>
<td>4</td>
<td>$n \log(n)$</td>
</tr>
<tr>
<td>5</td>
<td>$n$</td>
</tr>
<tr>
<td>6</td>
<td>$\log(n)$</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

### Diagram

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

The diagram illustrates the growth of the number of solution steps with respect to the problem size for different complexity categories.
Calculating Asymptotic Complexity

As \( n \) increases
- Highest complexity term dominates
- Can ignore lower complexity terms

Examples
- \( 2n + 100 \) \( \Rightarrow O(n) \)
- \( n \log(n) + 10n \) \( \Rightarrow O(n \log(n)) \)
- \( \frac{1}{2}n^2 + 100n \) \( \Rightarrow O(n^2) \)
- \( n^3 + 100n^2 \) \( \Rightarrow O(n^3) \)
- \( \frac{1}{100}2^n + 100n^4 \) \( \Rightarrow O(2^n) \)
Complexity Examples

2n + 100 ⇒ O(n)
Complexity Examples

\( \frac{1}{2} n \log(n) + 10 n \Rightarrow O(n\log(n)) \)
Complexity Examples

\[ \frac{1}{2} n^2 + 100 \, n \Rightarrow O(n^2) \]
 Complexity Examples

1/100 \(2^n + 100 \, n^4 \Rightarrow O(2^n)\)

---

The graph illustrates the growth of various functions compared to \(1/100 \, 2^n + 100 \, n^4\), showing why it is considered \(O(2^n)\).
Types of Case Analysis

- Can analyze different types (cases) of algorithm behavior

Types of analysis

- Best case
- Worst case
- Average case
- Amortized
Types of Case Analysis

Best case

- Smallest number of steps required
- Not very useful
- Example $\Rightarrow$ Find item in first place checked
Types of Case Analysis

- **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 pivot value from list
- Partition list into values smaller & bigger than pivot
- Recursively sort both lists
Quicksort Example

Quicksort properties

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

Can avoid worst-case behavior

- Select random pivot value
Types of Case Analysis

Average case

- Number of steps required for “typical” case
- Most useful metric in practice
- Different approaches
  - Average case
  - Expected case
Approaches to Average 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
Example problem

Average # of comparisons needed to find a number in the (sorted) array \( A[ ] = \{1, 4, 8, 12, 15\} \) using

- **Linear search**
  - Start from beginning, compare elements one at a time

- **Binary search**
  - Start from middle of array at index \( k \), compare element
  - If not element, repeat for top or bottom half of remaining array depending on whether element is smaller or greater than \( A[k] \)
Average Case : Linear Search

Algorithm

Find # of comparisons needed for each case

- 1 → 1 comparison (1)
- 4 → 2 comparisons (1, 4)
- 8 → 3 comparisons (1, 4, 8)
- 12 → 4 comparisons (1, 4, 8, 12)
- 15 → 5 comparisons (1, 4, 8, 12, 15)

Calc average = total # of comparisons / # cases

- Total # comparisons = 1 + 2 + 3 + 4 + 5 = 15
- # cases = 5
- Average = 3 comparisons / number
Average Case : Binary Search

Algorithm

- Find # of comparisons needed for each case
  - 1 → 3 comparisons (8, 4, 1)
  - 4 → 2 comparisons (8, 4)
  - 8 → 1 comparisons (8)
  - 12 → 2 comparisons (8, 12)
  - 15 → 3 comparisons (8, 12, 15)

- Calc average = total # of comparisons / # cases
  - Total # comparisons = 3 + 2 + 1 + 2 + 3 = 11
  - # cases = 5
  - Average = 2.2 comparisons / number
Average Case Example

Example problem 2

Average # of comparisons needed to find a number in a sorted array A[n] of size n using
- Linear search
- Binary search

For simplicity, we assume elements are stored in A[1] ... A[n]
Average Case : Linear Search

Algorithm

- Find # of comparisons needed for each case
  - ...

- Calc average = total # of comparisons / # cases
  - Total # comparisons = 1 + 2 + ... + n = \( \frac{1}{2} n^2 + 1 \)
  - # cases = n
  - Average ≈ \( \frac{1}{2} n \) comparisons / number
Average Case: Binary Search

Algorithm

Find # of comparisons needed for each case

- A[n/2] → 1 comp (A[n/2])
- ...

Calc average = total # of comparisons / # cases

- Total # comparisons = n/2 * log2(n) +
- n/4 * log2(n)–1 + ... + 1 = n log2(n)
- # cases = n
- Average ≈ log2(n) comparisons / number
Sample problem

- Given an array $a$ of integers
  - **find the subrange that has the maximum sum**
  - e.g., find low, high that maximizes $a[\text{low}] + a[\text{low}+1] + \ldots + a[\text{high}]$
  - only non empty ranges (low $\leq$ high)
  - If $a$ contained only nonnegative integers, would be low $= 0$, high $= a.length - 1$
  - but $a$ can contain negative numbers
  - Can assume that arithmetic overflow isn't an issue
public static int findBestRange(int [] a) {
    int bestSum = a[0];
    for(int low = 0; low < a.length; low++)
        for(int high = low; high < a.length; high++) {
            int sum = 0;
            for(int i = low; i <= high; i++) sum += a[i];
            if (bestSum < sum)
                bestSum = sum;
        }
    return bestSum;
}

// What is the complexity of the algorithm used here?
Can you find a better algorithm?