Algorithmic Complexity I

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 (usually too small)
    - Hardware
  - Affected by special cases (biased inputs)
  - Does not measure intrinsic efficiency
Asymptotic Analysis

- **Approach**
  - Mathematically analyze efficiency
  - Calculate time as function of input size $n$
    - $T = O[f(n)]$
    - $T$ is on the order of $f(n)$
    - “Big O” notation

- **Advantages**
  - Measures intrinsic efficiency
  - Dominates efficiency for large input sizes

---

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
1. Guess number = 1
2. If incorrect, increment guess by 1
3. Repeat until correct

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

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) = Linear time
Binary Search Algorithm

Algorithm

- Set $\Delta$ to $n/4$
- Guess number = $n/2$
- If too large, guess number – $\Delta$
- If too small, guess number + $\Delta$
- Reduce $\Delta$ by $1/2$
- Repeat until correct

Example

- Given number between 1…100
- Pick 20
- Guesses =
  - 50, $\Delta = 25$, Answer = too large, subtract $\Delta$
  - 25, $\Delta = 12$, Answer = too large, subtract $\Delta$
  - 13, $\Delta = 6$, Answer = too small, add $\Delta$
  - 19, $\Delta = 3$, Answer = too small, add $\Delta$
  - 22, $\Delta = 1$, Answer = too large, subtract $\Delta$
  - 21, $\Delta = 1$, Answer = too large, subtract $\Delta$
  - 20
- 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 = $O(\log_2(n)) = \text{Log time}$

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>

Run time increases linearly with input size.

For large values of n
- Time(2n) / 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>log₂(n)</td>
<td>5 * log₂(n) + 3</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

- log₂(n) and 5 * log₂(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

- Time(2n) – Time(n) approaches constant
- Base of logarithm does not matter
  - Simply a multiplicative factor
    \[ \log_a n = \left( \frac{\log_b n}{\log_b a} \right) \]
- Both are \( O(\log(n) ) \) programs
Asymptotic Complexity

Comparing two quadratic functions

<table>
<thead>
<tr>
<th>Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>n^2</td>
<td>2 n^2 + 8</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 2 n^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
- Time(2n) / 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

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

- $5n + 1000 \Rightarrow O(n)$
  - Select $M = 6, N_0 = 1000$
  - For $n \geq 1000$
    - $6n \geq 5n + 1000$ is always true
  - Example $\Rightarrow$ for $n = 1000$
    - $6000 \geq 5000 + 1000$

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
Complexity Category Example
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

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

\[ \frac{1}{2} n \log(n) + 10 n \Rightarrow O(n\log(n)) \]

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

- \( \frac{1}{100} 2^n + 100 n^4 \Rightarrow O(2^n) \)

![Complexity Examples Graph](image)

### 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 ⇒ 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 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 $\approx$ 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
Average Case Example

Example problem
- Average # of comparisons needed to find a number in the (sorted) array \( A[] = \{1, 4, 8, 12, 15\} \) using
  1. Linear search
     - Start from beginning, compare elements one at a time
  2. 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 \( A[k] \)

Algorithm
1. Find # of comparisons needed for each case
   - \( 1 \rightarrow 1 \) comparison (1)
   - \( 4 \rightarrow 2 \) comparisons (1, 4)
   - \( 8 \rightarrow 3 \) comparisons (1, 4, 8)
   - \( 12 \rightarrow 4 \) comparisons (1, 4, 8, 12)
   - \( 15 \rightarrow 5 \) comparisons (1, 4, 8, 12, 15)
2. 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**

1. **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)

2. **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**
  1. Linear search
  2. Binary search

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

Algorithm
1. Find # of comparisons needed for each case
   ...
2. Calc average = total # of comparisons / # cases
   - Total # comparisons = 1 + 2 + ... + n = \( \frac{1}{2} n^2 + 1 \)
   - # cases = n
   - Average \( \approx \frac{1}{2} n \) comparisons / number

Average Case : Binary Search

Algorithm
1. Find # of comparisons needed for each case
   - A[n/2] → 1 comp (A[n/2])
   ...
   - A[1], A[3]...A[n] → \( \log_2(n) \) comparisons
     (A[n/2], A[n/4], A[n/8]...A[1])
2. Calc average = total # of comparisons / # cases
   - Total # comparisons = \( n/2 \cdot \log_2(n) + 
     n/4 \cdot \log_2(n) - 1 + ... + 1 = n \log_2(n) \)
   - # cases = n
   - Average \( \approx \log_2(n) \) comparisons / number
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

Amortization Example

- Adding numbers to end of array of size \( k \)
  - If array is full, allocate new array
    - Allocation cost is \( O(\text{size of new array}) \)
    - Copy over contents of existing array

- Two approaches
  - Non-amortized
    - If array is full, allocate new array of size \( k+1 \)
  - Amortized
    - If array is full, allocate new array of size \( 2k \)
  - Compare their allocation cost
### Amortization Example

#### Non-amortized approach
- Allocation cost as table grows from 1..n

<table>
<thead>
<tr>
<th>Size (k)</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>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- **Total cost** ⇒ $n(n+1)/2$

#### Case analysis
- **Best case** ⇒ allocation cost = k
- **Worse case** ⇒ allocation cost = k
- **Amortized case** ⇒ allocation cost = $(n+1)/2$

---

### Amortization Example

#### Amortized approach
- Allocation cost as table grows from 1..n

<table>
<thead>
<tr>
<th>Size (k)</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>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Total cost** ⇒ $2(n – 1)$

#### Case analysis
- **Best case** ⇒ allocation cost = 0
- **Worse case** ⇒ allocation cost = $2(k – 1)$
- **Amortized case** ⇒ allocation cost = 2

- An individual step might take longer, but faster for any sequence of operations