

# Unit 3 Seminar

Principles of Computer Science and Algorithmic Thinking

## 1. Computational Thinking

Computational thinking is a problem-solving methodology that involves formulating problems and solutions in ways that can be effectively executed by an information-processing agent (human or computer).

### The Four Pillars of Computational Thinking

Pillar	Definition	Example
<b>Decomposition</b>	Breaking down complex problems into smaller, more manageable sub-problems	Building a website: design, frontend, backend, database, testing
<b>Pattern Recognition</b>	Identifying common structures, similarities, or trends within and across problems	Recognising that sorting emails is similar to sorting any list by criteria
<b>Abstraction</b>	Removing unnecessary details to focus on essential elements; generalising solutions	A map shows roads but omits individual trees and buildings
<b>Algorithmic Thinking</b>	Designing step-by-step instructions that can be followed to solve a problem	A recipe: precise steps to achieve a dish

### Activity: Designing an ATM Transaction Process

Applying computational thinking to ATM withdrawal:

#### Decomposition:

- Card insertion and validation
- PIN verification
- Transaction selection (withdraw, balance, transfer)
- Amount entry and validation
- Balance check
- Cash dispensing
- Receipt and card return

#### Pattern Recognition:

- All transactions follow: authenticate → select → validate → execute → confirm

#### Abstraction:

- Hide the complexity of network communication, encryption, and database operations from the user

### Algorithmic Thinking:

- IF balance >= requested\_amount THEN dispense cash ELSE display error

### Case Study: AI Recommendation Systems

Platforms like Netflix, Spotify, and Amazon use computational thinking to personalise content:

- **Decomposition:** User profiling, content analysis, similarity calculation, ranking, presentation
- **Pattern Recognition:** Users who liked X also liked Y (collaborative filtering)
- **Abstraction:** Represent users and items as vectors in feature space, ignoring irrelevant attributes
- **Algorithmic Thinking:** Matrix factorisation, nearest neighbour algorithms, neural networks for prediction

## 3. Algorithm Design and Problem-Solving Strategies

### What is an Algorithm?

An algorithm is a finite sequence of well-defined, unambiguous instructions that, given some input, produces an output and terminates in finite time.

### Properties of a Good Algorithm

- **Correctness:** Produces the right output for all valid inputs
- **Efficiency:** Uses minimal time and memory resources
- **Scalability:** Performance remains acceptable as input size grows
- **Clarity:** Easy to understand, implement, and maintain
- **Generality:** Solves a class of problems, not just one specific instance

### Comparing Algorithmic Strategies

Strategy	How It Works	Pros	Cons
<b>Brute Force</b>	Try all possible solutions and select the correct one	Simple to implement; guaranteed to find a solution if one exists	Very slow for large inputs; often impractical
<b>Divide and Conquer</b>	Split the problem into smaller subproblems, solve each, and combine results	Efficient; often $O(n \log n)$ ; parallelisable	Overhead from recursion; not suitable for all problems
<b>Greedy</b>	Make a locally optimal choice at each step, hoping for a global optimum	Fast, simple, works well for certain problems	May not find optimal solution; short-sighted
<b>Dynamic Programming</b>	Break into overlapping	Optimal solutions; avoid redundant computation	Higher memory usage; complex to design

	subproblems; store and reuse solutions		
--	--	--	--

## Recursion vs Iteration

Aspect	Recursion	Iteration
<b>Definition</b>	Function calls itself with a smaller input	Loops (for, while) repeat until condition met
<b>Memory</b>	Uses call stack (risk of stack overflow)	Constant memory for loop variables
<b>Best For</b>	Tree traversal, divide and conquer, and naturally recursive problems	Simple loops, performance-critical code
<b>Example</b>	$\text{factorial}(n) = n \times \text{factorial}(n-1)$	<code>result = 1; for i in 1..n: result *= i</code>

## Activity: Linear Search vs Binary Search

Aspect	Linear Search	Binary Search
<b>How It Works</b>	Check each element from start to end	Check middle; eliminate half each time
<b>Prerequisite</b>	None (works on unsorted data)	Data must be sorted
<b>Time Complexity</b>	$O(n)$ - linear	$O(\log n)$ - logarithmic
<b>1,000 items (worst)</b>	1,000 comparisons	10 comparisons
<b>1,000,000 items</b>	1,000,000 comparisons	20 comparisons
<b>Best Use Case</b>	Small datasets; unsorted data; single search	Large sorted datasets; frequent searches

## Example: How Google Search Optimises Algorithms

- **Inverted Index:** Pre-built data structure mapping terms to documents (enables fast lookup)
- **PageRank:** Greedy/iterative algorithm ranking pages by link authority
- **Caching:** Store results of common queries (dynamic programming principle)
- **Distributed Computing:** Divide and conquer across thousands of servers

## 4. Time and Space Complexity – Big-O Notation

### What is Big-O Notation?

Big-O notation describes the upper bound of an algorithm's growth rate as input size increases. It answers the question: "How does performance scale?"

## Why It Matters

- Predicts performance for large datasets before implementation
- Enables comparison between algorithms independent of hardware
- Critical for systems handling millions/billions of operations
- Helps identify bottlenecks and optimisation opportunities

## Common Time Complexities

Notation	Name	Description	Example
$O(1)$	Constant	Same time, regardless of input size	Array access by index; hash table lookup
$O(\log n)$	Logarithmic	Halves problem with each step	Binary search
$O(n)$	Linear	Time grows directly with input	Linear search; single loop
$O(n \log n)$	Linearithmic	Efficient divide and conquer	Merge sort; QuickSort (average)
$O(n^2)$	Quadratic	Time squares with input; nested loops	Bubble sort: comparing all pairs
$O(2^n)$	Exponential	Doubles with each additional input	Recursive Fibonacci; brute force subsets

## Practical Impact: Operations for $n = 1,000,000$

Complexity	Operations
$O(1)$	1
$O(\log n)$	20
$O(n)$	1,000,000
$O(n \log n)$	20,000,000
$O(n^2)$	1,000,000,000,000 (1 trillion)

## Space Complexity

Space complexity measures the additional memory an algorithm needs beyond the input:

- **$O(1)$  space:** In-place algorithms using only a few variables (e.g., bubble sort)
- **$O(n)$  space:** Requires additional storage proportional to input (e.g., merge sort)
- **Trade-off:** Often must choose between time efficiency and space efficiency

## Activity: Sorting Algorithm Comparison

Algorithm	Best	Average	Worst	Space
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$ - in-place
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$ - needs extra array

### Bubble Sort

- **How:** Repeatedly swap adjacent elements if in wrong order; "bubbles" largest to end
- **When to use:** Educational purposes only; rarely used in practice due to  $O(n^2)$  performance

### QuickSort

- **How:** Pick pivot; partition into smaller/larger; recursively sort partitions
- **When to use:** General purpose; fast in practice; used in many standard libraries

### Merge Sort

- **How:** Divide the array in half; recursively sort each half; merge the sorted halves
- **When to use:** When guaranteed  $O(n \log n)$  needed; stable sort required; external sorting (files)

## Case Study: Social Media News Feed Optimisation

Facebook and Twitter handle billions of posts. Efficiency is critical:

- **Challenge:** Sort and rank thousands of posts for each user in real-time
- **Data structures:** Hash tables  $O(1)$ , priority queues  $O(\log n)$ , graphs for social connections
- **Caching:** Pre-computed feeds stored in memory; reduces computation
- **Trade-offs:** Balance between freshness (more computation) and speed (more caching)

## 5. Quick Reference: Key Formulas and Concepts

- **Algorithm:** Finite sequence of unambiguous instructions producing output from input
- **Big-O:** Upper bound of growth rate; describes worst-case scalability
- **$\log_2(n)$ :** How many times can you halve  $n$  before reaching 1 (e.g.,  $\log_2(1024) = 10$ )
- **Decomposition:** Break complex → simple sub-problems
- **Abstraction:** Hide details, focus on essentials
- **Divide and Conquer:** Split → solve → combine (typically  $O(n \log n)$ )
- **Greedy:** Local optimum at each step; fast but not always optimal
- **Binary vs Linear Search:**  $O(\log n)$  vs  $O(n)$ ; binary requires sorted data