Computer Science(UX Design) -Lecture 1

1. What Computer Science *Actually* Is

Computer Science (CS)

→ The study of **computational systems**
→ How problems are **modeled**, **solved**, and **automated** using computers

Core idea

- Computers don't "think" like humans
- They **follow logic and instructions**
- CS teaches us **how to design those instructions**

Reality vs theory

- **On paper**: Theory → Development → Deployment
- **In practice**: Continuous improvement, updates, optimization

Every update on your phone is computer science in motion.

2. Why Computer Science Matters

Key advantages of computers

1. **Automation**
   - Repetitive tasks handled instantly
   - Example: billing, payroll, checkout systems
2. **Speed**
   - Massive computations done in seconds
   - Searching the internet vs manual lookup
3. **Accuracy**
   - Eliminates human fatigue errors
   - Caveat: **Garbage In → Garbage Out**
4. **Scalability**
   - One system can serve millions simultaneously

3. Core Terminology (You'll See These Forever)

Programming

- Designing **instructions** a computer can follow

Coding

- Writing those instructions in **specific languages**

Programming Language

- A formal system for communicating with computers
- Example: Python, Java, C, etc.

4. Mathematics & Computing Relationship

- Nature itself is **mathematical**
- Computers model:
  - Motion, Patterns, Proportions, Geometry, Probability

Examples

- Flight simulation, Facial recognition, Physics engines, Weather forecasting, Biomechanics

This is why pilots can train **without flying**.

5. Major Fields of Computer Science

Foundational Areas

- Artificial Intelligence (AI), Computer Systems & Networks, Database Systems, Software Engineering, Human-Computer Interaction (HCI), Computer Vision & Graphics, Bioinformatics, Theory of Computation (algorithms & complexity)

Key insight

- You **don't learn everything**
- You **specialize deeply** in one or two

6. Historical Evolution of Computers

Early Computing

- **Abacus** – simple calculations (2000+ years ago)

Mechanical Era

- **Charles Babbage**
  - Built the **Difference Engine**
  - Introduced:
    - Storage
    - Mechanical computation
    - Output mechanisms

First General-Purpose Computer

- **ENIAC**
  - 30 tons, 18,000 vacuum tubes
  - 160 kW power, Programmable
  - Used for:
    - Weather
    - Physics, Military calculations

Architecture Breakthrough

- **John von Neumann**
  - Introduced **stored-program architecture**
  - Foundation of all modern computers

Transistor & IC Era

- Transistors replaced vacuum tubes
- Integrated Circuits (ICs):
  - Smaller
  - Faster
  - More reliable

Programming Evolution

- Assembly → High-level languages
- COBOL for business
- Operating systems introduced

7. Generations of Computers (High Level)

1. Vacuum tubes
2. Transistors
3. Integrated circuits
4. Microprocessors
5. Modern computing (parallelism, graphics, AI)

8. Artificial Intelligence

- Definition: Programming machines to **mimic human intelligence**

- Foundational Figure: **Alan Turing**
- Turing Test: If a machine fools humans ≥30% of the time in conversation → considered intelligent
- Notable Event: 2014: Eugene Goostman chatbot claimed to pass Turing Test

9. Machine Learning (ML)

What it is

- Systems that **learn from data**
- Not explicitly programmed for every rule

Core components

- Data, Model, Training, Prediction

Applications

- Spam filtering, Search engines, OCR, Fraud detection, Computer vision

10. Machine Learning Models

Artificial Neural Networks

- Inspired by the human brain
- Nodes = neurons
- Connections = synapses

Deep Learning

- Neural networks with **many layers**
- Used in: Vision, Speech recognition

Genetic Algorithms

- Inspired by natural selection
- Uses: Mutation, Crossover, Fitness evaluation

Decision Trees

- Tree-based logic
- Branches = conditions
- Leaves = outcomes

Federated Learning

- Learning from **distributed data**
- Example: keyboard prediction without uploading personal data

11. Internet of Things (IoT)

- Definition: Interconnection of physical devices with intelligence

Examples

- Smart speakers, Smart watches, Health monitors, Home automation systems

12. Quantum Computing

- Based on **quantum mechanics**
- Uses qubits instead of bits
- Potential to outperform classical computers massively
- Currently: Mostly theoretical, Some working prototypes exist

13. Real-World Applications of Computing

Business & Finance

- Banking systems, Stock markets, Payroll, Budgeting, Financial analytics

Communication

- Digital calls, Messaging platforms, Collaboration tools

Manufacturing

- Robotics, Precision automation

Consumer Electronics

- Smartphones, Smart TVs, Wearables, AI chips in devices

Healthcare & Accessibility

- Assistive technologies, Brain-controlled prosthetics, Medical simulations

Science & Research

- Weather prediction, Space exploration, Molecular modeling, Physics simulations

14. Career & Opportunity Outlook

- Web development, Software engineering, Data analysis, Mobile development, Systems design

Demand for computing skills is **accelerating**, not slowing.

15. Core Takeaway

- Computer science = problem-solving at scale
- Data is the fuel
- Algorithms are the logic
- Computers amplify human capability
- The field has no fixed ceiling

Computer Science(UX Design) -Lecture 2

1. Information Processing (Core Concept)

Definition

- **Data** → raw facts
- **Information** → processed data presented meaningfully

Conceptual analogy

- **Cake baking**
  - Ingredients = data
  - Recipe = program
  - Baker = CPU
  - Cake = information

2. Information Processing Cycle

2.1 Data Collection

- First step of processing
- Data sources:
  - Environment (via sensors)
  - Input devices (keyboard, mouse, touch, camera)
  - Files & databases
  - Previously processed data
  - Computer-generated data

Key idea

- Data must be in a **usable format**
- Raw data is **converted to machine-readable form**

2.2 Data Processing

- Done by **software programs**
- Executed by **CPU**
- Programs = instructions that define:
  - What data to use
  - Order of operations
  - Duration & conditions

Performance depends on:

- Processor speed
- Bus size
- Cache size
- Architecture

2.3 Storage

Temporary Storage (Memory)

- **RAM (Random Access Memory)**
- Volatile → cleared when power is off
- Stores currently used data & programs

Permanent Storage

- Drives (HDD, SSD)
- Flash storage
- Phone storage
- Retains data after shutdown

Storage formats

- Files, Databases

2.4 Output (Information Presentation)

- Output forms: Visual (screens, print), Audio (sound, music), Physical actions (robots, machines)

3. Operating System (OS)

3.1 Definition

An **Operating System** is a collection of procedures that:

- Allows **multiple users** to share computing resources
- Manages **hardware & software coordination**

3.2 Core Role of OS

- Acts as **middle layer** between:
    - Hardware, Applications, Users

4. Boot Process (System Startup)

Key terms

- **Boot** = starting the system
- Origin: "pulling oneself up by bootstraps"

Boot sequence

1. Power ON
2. **BIOS** (Basic Input Output System)
    - Initializes hardware
    - Performs checks
3. **Boot Loader**
    - Locates OS
    - Loads kernel into RAM
4. **Kernel takes control**
5. Drivers, services, applications loaded
6. Runtime state reached (system usable)

Rebooting

- **Hard reboot**: power cut
- **Soft reboot**: RAM cleared, power maintained

5. Kernel (Heart of the OS)

Definition:

- Core component of OS
- Always resident in memory
- Full control over system resources

Responsibilities

- Memory control
- Process scheduling
- Device communication
- Security enforcement

6. OS Functional Responsibilities

1. Memory Management
2. Processor Scheduling
3. Device Management (via drivers)
4. File Management
5. Security & Permissions
6. Resource Allocation
7. Error Handling
8. Application Platform

7. OS Architecture Layers

- Hardware, Kernel, Shell (interface), Utilities & Applications

8. Kernel Types

8.1 Monolithic Kernel

- Kernel & services in same space
- Fast execution
- Larger OS
- Example: **Linux**

8.2 Microkernel

- Minimal kernel
- Services run in user space
- Modular, secure
- Message-based communication

8.3 Hybrid Kernel

- Combines monolithic speed + microkernel modularity
- Example: **Microsoft Windows**

8.4 Nanokernel

- Extremely small
- Hardware abstraction only

8.5 Exokernel

- Minimal abstractions
- Application-specific customization

9. Kernel Space vs User Space

- **Kernel Space**
  - Privileged
  - Direct hardware access
- **User Space**
  - Applications
  - Isolated for safety

Purpose: prevent total system failure if an app crashes.

10. Device Drivers

What they are

- Software layer between hardware & OS
- Abstract hardware as files

Why they exist

- Prevent each app from managing hardware itself
- Improve efficiency & compatibility

Example

- Brightness change:
  - OS writes value → driver
  - Driver translates to hardware signal

11. Operating System Categories

- Three major families: Unix-like systems, Linux, Windows

12. Unix-Like Systems

Types:

1. **Genetic Unix** – original codebase
2. **Branded Unix** – commercial variants
3. **Functional Unix** – Unix-like behavior

13. macOS

- Unix-derived (BSD roots)
- Developed by **Apple Inc.**
- Written in:
  - C, C++, Objective-C, Swift
- Known for:
  - Stability, Security, Performance

14. Free Software Movement

Key Figure

- **Richard Stallman**

Contributions

- GNU Project (1983)
- Free Software Foundation (1985)
- GNU General Public License (GPL)

15. Linux

Characteristics

- Open source
- Kernel + distributions
- Highly customizable

Distributions

- Desktop: Linux Mint, Ubuntu
- Lightweight: Puppy Linux
- Enterprise: Red Hat, SUSE

Desktop environments

- GNOME, KDE, XFCE, MATE

Usage

- ~96% of servers
- Supercomputers
- Embedded systems
- IoT devices

16. Android

- Based on Linux kernel
- Developed by **Google**
- Uses:
  - Linux Kernel
  - Android Runtime (ART)
- Runs apps inside a **virtual machine**
- Requires more RAM due to abstraction layers

Market impact

- ~2 billion users
- ~85% smartphone market share
- Used in phones, cars, TVs, consoles, cameras

17. Virtual Machines

- Definition: OS running on top of another OS

Terms

- Host OS → hardware access
- Guest OS → virtualized environment

Trade-off

- Compatibility ↑
- Performance ↓

18. Windows

- Proprietary OS by **Microsoft**
- Written in: C (kernel), C++, C#
- Market share: ~77% desktop OS
- Closed source
- Apps typically released here first

19. APIs (Application Programming Interfaces)

Purpose

- Allow developers to access OS functionality
- Avoid rewriting low-level code

Importance

- Critical in proprietary OS
- Central to Android development

20. Why This Matters to Programmers

- OS = code written by programmers
- Understanding OS internals helps:
  - Choose correct platform
  - Avoid unsupported designs
  - Write efficient code
  - Prevent wasted time & cost

21. Core Takeaway

- Information processing = **input → process → store → output**
- OS is the **most important software**
- Kernel is the **brain**
- Drivers are the **translators**
- APIs are the **bridges**
- Knowing OS internals = **better engineering decisions**

Computer Science(UX Design) -Lecture 3

1. Computing Systems — Overview

A **computer system** is built using a predefined architecture, just like a house follows a blueprint.

Core components (present in all computers)

- **Input devices** – feed raw data
- **CPU** – processes data
- **Storage devices** – store data & programs
- **Output devices** – present results

2. Central Processing Unit (CPU)

The CPU is a **small chip**, not the computer cabinet.

Internal components of the CPU

2.1 Arithmetic Logic Unit (ALU)

- Performs:
  - Arithmetic operations (add, subtract, etc.)
  - Logical operations (AND, OR, comparisons)
- Handles **decision making**

2.2 Registers

- Very small, ultra-fast memory inside CPU
- Types:
  - **General-purpose registers**
  - **Special-purpose registers**
- Faster than cache and RAM

2.3 Cache

- High-speed memory inside the processor
- Stores:
  - Frequently used data
  - Recently used instructions
- Purpose: reduce access time to RAM
- Measured in MB (e.g., 3 MB cache)

2.4 Buses

High-speed communication pathways

- **Address bus** → carries memory addresses
- **Data bus** → carries actual data
- **Control bus** → carries control signals

2.5 Clock

- Synchronizes all CPU operations
- Measured in **Hertz (Hz)**
- Example:
  - 2.5 GHz = 2.5 billion cycles/second
- One cycle ≈ one instruction step

3. Von Neumann Architecture

Proposed in 1945 by **John von Neumann**

Core idea

- **Programs and data are stored together in memory**
- Enables **general-purpose computing**

Why it mattered

- Eliminated hard-wiring for each task
- Made computers programmable via software

3.1 Special Registers in Von Neumann Architecture

Register	Purpose
PC (Program Counter)	Holds next instruction address
CIR (Current Instruction Register)	Holds instruction being executed
MAR (Memory Address Register)	Holds memory address to access
MDR (Memory Data Register)	Holds data being transferred
ACC (Accumulator)	Stores intermediate & final results

4. Instruction Sets

Instruction Set

- Complete set of commands a CPU understands
- Controls transistor switching

4.1 CISC vs RISC

CISC – Complex Instruction Set Computer

- Goal: fewer instructions per program
- Uses **microcode**
- Instructions span multiple cycles
- Hardware complexity: high
- Pros:
  - Uses less RAM
  - Flexible instruction expansion

RISC – Reduced Instruction Set Computer

- Simple instructions

- One instruction per clock cycle
- Enables **pipelining**
- More registers, fewer transistors
- Pros:
  - Predictable performance
  - Easier compiler optimization
- Cons:
  - Requires more RAM

Analogy

- **CISC** → adult given one complex task
- **RISC** → child guided step-by-step

5. Digital Computing Fundamentals

Binary System

- Uses only:**0** (off), **1** (on)
- Based on transistor states

Bit:

- Single binary digit (0 or 1)

5.1 Byte & Data Units

- **Nibble** = 4 bits
- **Byte** = 8 bits (standard)
- **Word** = 16 bits

6. Memory Addressing & Powers of Two

- Memory addresses are binary
- Adding one address line → doubles capacity

Example

- 4 address lines → 2^4 = 16 locations
- 5 address lines → 32 locations

Why powers of two?

- Prevent unused addresses
- Reduce controller complexity
- Avoid data loss

Powers of two became a **de facto standard**

7. Limits of Computing

7.1 Where Humans Are Better

- Emotional understanding
- Intuition
- Contextual decisions
- Creativity

7.2 Affective Computing

- AI branch focused on:
  - Emotion recognition
  - Facial expressions
  - Human emotional context

7.3 Creativity in AI

- **Inceptionism**
  - AI generates images from noise
- Example: Google "dreaming" AI, Pig-snails, camel-birds

7.4 Self-Improvement Limitation

- Computers don't evolve autonomously
- Improvements require:
  - Engineers, Programmers
- Exception: learning systems

8. Learning Systems

Example

- **AlphaGo (DeepMind)**
  - Learned games autonomously
  - Improved through iteration

Still dependent on initial human-designed frameworks

9. Decision Making in Computers

- All decisions reduce to:
  - **True / False**
- Based on provided data & logic
- No genuine independent thought

9.1 Natural Language Challenges

- Accents
- Pronunciation variation
- Context
- Figurative language

9.2 Fuzzy Logic

- Allows partial truth values
- Avoids strict true/false boundaries
- Helps with:
  - Speech recognition
  - Pattern ambiguity

10. Fetch–Decode–Execute Cycle

CPU operation loop

- F**etch** instruction from RAM > **Decode** instruction > **Execute** instruction
- Synchronized by clock
- One complete loop = one instruction cycle

11. Factors Affecting Computer Speed

1. Clock Speed
   - More cycles per second → faster CPU
2. Cache Size
   - Larger cache → faster access
3. Number of Cores
   - Multiple processing units
   - Common in powers of two
   - Quad-core ≫ dual-core

12. Core Takeaway

- Computers are **deterministic machines**
- Everything reduces to:
  - Binary states, Timed instruction cycles
- Power comes from:
  - Architecture, Instruction design, Parallelism
- Limits remain in:
  - Emotion, Creativity, Contextual reasoning

Computer Science(UX Design) -Lecture 4

1. Natural Language vs Computer Language

Natural Language

- Refers to **human language**
- Used to express:
  - Thoughts
  - Identity
  - Emotions
  - Imagination
- Logical **and** emotional
- Highly **ambiguous**

Ambiguity

- Same sentence → different meanings based on:
  - Emphasis, Tone, Context
- Useful for humans, **problematic for computers**

Computer / Programming Language

- A **special-purpose language**
- Used to give **precise instructions**
- Must be:Unambiguous, Deterministic ,Strictly structured

If code is confusing to read, it is badly written code.

2. Language Structure (Borrowed into Programming)

Syntax

- Order of words / symbols, Rules of arrangement

Semantics

- Meaning of words / symbols

In simple terms:

Syntax = structure

Semantics = meaning

Programming languages borrow **syntax heavily** but minimize semantics to avoid ambiguity.

3. Why Programming Languages Exist

- Early computers used **pure binary (1s and 0s)**
- Extremely error-prone
- Impossible to scale

Purpose of programming languages

- Make computers usable
- Eliminate rewiring
- Express logic clearly
- Abstract hardware complexity

4. Binary Representation of Information

Binary System

- Based on: **0** (off), **1** (on)
- Fundamental to all computing

Bit

- Binary digit
- Smallest unit of information

Bytes & Units

- **Nibble** = 4 bits
- **Byte** = 8 bits (standard)
- **Word** = CPU-dependent (32-bit / 64-bit)

5. Character Encoding

Unicode

Designed to represent **all human languages**

UTF (Unicode Transformation Format)

- **UTF-7** – legacy email compatibility
- **UTF-8** – most popular
  - Variable width (8–48 bits)
  - Backward compatible with ASCII
- **UTF-16** – 16–32 bits
- **UTF-32** – fixed 32 bits

Capacity

- Characters = 2^n (n = number of bits)
- UTF-32 → ~4.2 billion characters

ASCII

- 7-bit encoding
- 2^7 = **128 characters**
- English-centric
- Counting starts at **0**

Limit:

Cannot represent global languages.

6. Parity Bits (Error Detection)

Used to check **data integrity**

- Even Parity: Total number of 1s must be even
- Odd Parity: Total number of 1s must be odd

Parity bit is adjusted accordingly.

7. Machine Language & Instruction Sets

Machine Language

- Actual 1s and 0s executed by hardware
- All programming languages compile/translate into this

Instruction Set Architecture (ISA)

- Programmer-visible CPU design
- Defines:
  - Supported operations
  - Instruction formats
  - Word size
- Example: **x86**, **ARM**

Word Size

- Amount of data CPU processes at once
- Common sizes:
  - 32-bit, 64-bit

Compatibility

- 32-bit programs → run on 64-bit systems
- 64-bit programs → □ on 32-bit systems

8. Computer Instructions

Each instruction has **three parts**:

1. **Opcode:** Operation to perform
2. **Address Field:** Memory/register location
3. **Mode Field:** How operand/address is interpreted

Instruction Types: Memory reference, Register reference, Input/Output

9. Transducers (Input Conversion)

- Purpose: Convert environmental data → digital signals

Examples

- Microphone, Accelerometer, Pressure sensor, Hall sensor, Display, Motors

Smartphones are packed with transducers.

10. Programming Languages (Conceptual View)

- Describe tasks **step-by-step**
- Follow strict rules
- Require understanding of:
  - Hardware, OS, Architecture

Before building a computer:

- An **abstract machine** is designed

11. Abstract Computational Models

These are **theoretical models**, not real machines.

11.1 Finite State Machines (FSM)

- Limited memory
- Generates **regular languages**
- Used for:
  - Simple logic
  - Pattern matching
  - Regular expressions

11.2 Pushdown Automata (PDA)

- FSM + **stack**
- Stack = LIFO (Last In, First Out)
- Accepts **context-free languages**
- Used in:
  - Parsers, Compilers, Matching parentheses

11.3 Linear Bounded Automata (LBA)

- PDA + finite tape
- Accepts **context-sensitive languages**

11.4 Turing Machine

Invented by **Alan Turing** (1936)

Characteristics

- Infinite tape (memory), Read/write head, Transition table
- Can:
  - Halt, Run forever

Importance

- Most powerful computational model, Defines limits of computability

Any real computer can be simulated by a Turing Machine (given enough memory).

12. Decidability

- **Decidable language**: TM halts for all inputs
- **Recognizable language**: TM may not halt for invalid inputs

All decidable languages are recognizable, but not vice versa.

13. Chomsky Hierarchy

Proposed by **Noam Chomsky** (1956)

Language hierarchy (from weakest → strongest)

	Type 3	**Type 2**	**Type 1**	**Type 0**
Grammar	Regular	Context-Free	Context-Sensitive	Unrestricted
Machine	Finite Automata	Pushdown Automata	Linear Bounded Automata	Turing Machine

Each level is a **subset of the next**.

14. Core Takeaway

- Human language → expressive but ambiguous
- Computer language → strict, deterministic
- Binary underpins everything
- Encoding enables global communication
- Instruction sets bridge software & hardware
- Abstract machines define **what is computable**
- Chomsky hierarchy classifies **language power**

Computer Science(UX Design) -Lecture 5

1. Brief History of Programming Languages

Early Mechanical Roots

- **Jacquard Loom (early 1800s)**
  - Used punch cards to control fabric patterns
  - First example of **machine instruction via symbolic input**

First Programmer

- **Ada Lovelace**
  - Wrote an algorithm for **Charles Babbage**'s Analytical Engine
  - Target problem: Bernoulli numbers
  - October 15 celebrated as **Ada Lovelace Day**

Stored-Program Era (1940s)

- Emergence of programmable electronic computers
- **Alec Glennie**
  - Developed **Autocode** for the Mark I
  - First high-level programming language concept

Assembly Language

- Invented by **Kathleen Booth**
- Replaced raw binary with mnemonics (ADD, SUB, MUL)
- Still used today for:
  - Device drivers
  - Embedded systems
  - Real-time systems

1960s–1970s

- Explosion of programming languages
- Evolution of:
  - C → C++
  - Modular programming
  - Object-oriented concepts

Internet Era

- Shift toward:
  - Programmer productivity
  - Rapid development
- Rise of:
  - Scripting languages
  - Interpreted languages

2. What Is a Programming Language?

Definition: A **systematic method** of describing a computational process using:

- Arithmetic, Logic, Control flow, Input / Output

3. Programming Paradigms (Models)

There are **two primary paradigms**:

3.1 Imperative Programming

Core idea

- Focuses on **how** a task is done, Program state changes step by step

Characteristics

- Assignment statements, Global state mutation, Close to machine architecture

Limitations

- Poor parallelism, Complex state management

Imperative Subtypes

Procedural Programming

- Sequential execution
- Heavy use of loops & variables
- Examples:
  - C, C++, Java, Pascal

Object-Oriented Programming (OOP)

- System modeled as interacting objects
- Key concepts: Encapsulation, Inheritance, Polymorphism
- Emphasizes reusability

Parallel Processing

- Divide-and-conquer strategy, Tasks distributed across processors

3.2 Declarative Programming

Core idea

- Focuses on **what** needs to be done, Not how it is achieved

Characteristics

- No explicit loops, No variable reassignment, High-level abstraction

Declarative Subtypes

Logic Programming

- Program = facts + rules
- Computer reasons about consequences
- Example: Prolog

Functional Programming

- Pure mathematical functions
- Immutable data
- Recursion instead of loops
- Functions passed as values
- Example: **ReactJS**

Database Programming

- Data-driven logic
- Queries describe desired result
- Uses **SQL**
- Managed by **DBMS**
- Strong data consistency guarantees

4. Low-Level vs High-Level Languages

Low-Level Languages

- Hardware-specific
- Assembly, machine code
- Fast but non-portable

High-Level Languages

- Hardware-independent, Portable source code
- Compiled per target architecture
- Large libraries & abstractions

Emulator

- Software that mimics another architecture, Inefficient and resource-heavy
- Avoided when high-level languages are available

5. Translation of Programs

All programs must become **machine language** to run.

5.1 Translation Tools

Assembler

- Assembly → machine code

Compiler

- High-level code → machine code
- Produces executable file
- Target-specific

Interpreter

- Translates line by line
- Executes immediately
- No standalone executable

6. Compilation Process (High-Level)

Phase 1: Preprocessing

- Handles macros, includes

Phase 2: Analysis

Lexical Analysis (Scanner)

- Converts code into tokens
- Removes whitespace/comments
- Detects invalid symbols

Syntax Analysis (Parser)

- Checks grammar rules
- Builds parse tree
- Reports syntax errors

Semantic Analysis

- Checks logical meaning
- Undeclared variables
- Type mismatches

Symbol Table

- Stores:
  - Variables, Functions, Classes
- Used across compiler phases

Phase 3: Intermediate Code Generation

- Architecture-independent
- Assembly-like representation

Phase 4: Code Optimization

- Improves speed
- Reduces size

Phase 5: Code Generation

- Translates to target machine code
- Output: **target program**

7. Assembly, Linking, and Loading

Assembler (Two Passes)

Pass 1

- Determines memory requirements
- Builds assembler symbol table

Pass 2

- Produces **relocatable machine code**

Linker

- Combines multiple object files
- Resolves references
- Produces single executable

Loader

- Loads executable into memory
- Starts execution
- Fails if references unresolved

8. Machine Code Types

- **Relocatable Machine Code:** Can load at different memory locations

- **Absolute Machine Code**
  - Final executable
  - Hardware-specific
  - Non-portable

9. Portability Implications

- Source code → portable
- Compiled executable → NOT portable
- Must compile separately for:
  - x86, ARM, Different operating systems

Example:

- Same Microsoft Office source
- Different binaries for Windows, macOS, Android

10. Core Takeaways

- Programming languages evolved with hardware
- Paradigms shape how problems are solved
- High-level languages trade control for productivity
- Compilation is a **multi-stage pipeline**
- Executables are architecture-specific
- Understanding this pipeline = better debugging & design

Computer Science(UX Design) -Lecture 6

1. Problem Formulation in Computer Science
Why problem formulation matters
- Before writing code, you must know **why** the program exists
- Prevents: Missing requirements, Incomplete solutions, Poor design decisions

Problem formulation
- Translating a **real-world need** into a **computationally solvable form**
- The programmer's responsibility is to:
    - Clarify ambiguity, Identify constraints, Define goals precisely

2. What Is a Computing Problem?
- In computer science, a **problem** is:
    - A task or a set of related tasks, That may be solvable by a computer

Key requirement
- Problems must be **explicitly stated**
- Computers cannot handle:
    - Vague questions, Emotional concepts
- Ambiguous goals: (e.g., "What is joy?")

3. Thinking Computationally
A computer scientist must:
- Break problems into smaller parts
- Identify what **can** and **cannot** be computed
- Decide if a problem is:
    - Solvable, Partially solvable, Unsolvable

4. Five Main Types of Computational Problems
4.1 Decision Problems
- Output is **YES or NO**
- Tests whether a property holds

Example: Is X + Y = Z?
Properties
- Closely related to function problems
- Problems can be:
    - **Decidable, Partially decidable, Undecidable**

Undecidable problems may **run forever** without producing an answer.

4.2 Search Problems
- Goal: **find a solution**, not just confirm existence
- Always associated with a decision problem

Defined by:
- Set of states, Start state, Goal state, Successor function, Goal test function

Example: Searching for a number in a list

- Search problems: Answer **where / how**, Not just **whether**

4.3 Counting Problems

- Goal: **count the number of valid solutions**
- Example: Number of perfect matchings in a graph
- Challenges: Brute-force search becomes impractical
- Requires clever algorithms (e.g., Edmonds' algorithm)

4.4 Optimization Problems

- Goal: find the **best solution**
- "Best" = **minimum cost / weight / time / distance**
- Weight: Represents resource usage

Examples: Route planning (maps), Airline scheduling, **Dijkstra's shortest path**
Types: Continuous optimization, Discrete optimization

4.5 Function Problems

- Every input has a corresponding output
- Output is **not just YES/NO**

Example: Mathematical functions (x / y)
Relationship

- Every function problem → decision problem
- Decision problem = graph of the function

5. General Characteristics of Computational Problems

Defined by: A **task**, A set of **input instances**

- Same task, different inputs
- Example: Addition is always the same process, Only numbers change

6. Tractable vs Intractable Problems

Tractable Problems

- Solvable: Algorithmically, In **reasonable (polynomial) time**

Intractable Problems

- Solvable only with:
    - Exponential or worse time complexity
- Impractical for large inputs
- Examples: Traveling Salesman Problem (TSP), School timetabling

Traveling Salesman Problem (TSP)

- Find shortest route visiting all cities once
- Cost = distance + time + resources

Reality: Exact solutions possible only for limited sizes
Record (2006): ~85,900 cities
Real-world use: PCB drilling paths, Manufacturing optimization

7. Approximate & Heuristic Solutions

Suboptimal Solutions

- Not perfect, Good enough, Solvable in reasonable time

Heuristic Algorithms

- Use: Experience, Rules of thumb, Judgment
- Avoid exhaustive search

Example: Always pick cheapest next route (greedy approach)

8. Polynomial Time vs Exponential Time

- **Polynomial time (P)**: Considered “fast”, Scales reasonably with input size
- **Exponential time**: Grows too quickly, Becomes unusable

9. Unsolvable / Undecidable Problems

- Definition: No algorithm can solve the problem for all inputs
- Famous example: **Halting Problem**
- Question: Given a program and input, can we know if it will halt or run forever without running it?
- Result: Proven undecidable by **Alan Turing**

10. Mathematics and Computer Science

Why math is essential

- Computer science is built on:
    - Set theory, Graph theory, Probability, Number theory

Mathematics as a language

- Precise, Unambiguous, Universal

Computer science is often considered a **subset of mathematical sciences**.

11. Mathematical Modeling

Definition

- Translating real-world problems into mathematical form
- Enables: Prediction, Simulation, Optimization
- Benefits: Reveals hidden relationships, Allows controlled experimentation

Types of Models

Mechanistic Models

- Based on physical laws
- Detailed, theory-heavy

Empirical Models

- Based on observed data
- Respond to changing conditions

Stochastic Models

- Probabilistic
- Predict distributions

Deterministic Models

- Same input → same output always

12. Dining Philosophers Problem

- Classic synchronization problem
- Models: Resource sharing, Deadlock prevention

Real-world analogy

- **Multithreading**, CPU time sharing, Device access coordination

13. Writing Problems Clearly

A well-formulated problem must be:

1. **Precise** (unambiguous)
2. Have a **clear initial state**
3. Have a **clear goal state**
4. Define **rules & constraints**
5. Include **all components**

Example: Water Jug Problem

- Jugs: 4L and 3L
- Goal: exactly 2L in 4L jug
- No measuring markers
- Clearly defined states & rules

14. Final Takeaways

- Not all problems are computable
- Not all solvable problems are practical
- Problem formulation determines:
  - Algorithm choice, Feasibility, Performance
- Mathematics provides the structure
- Heuristics make the impossible workable

Computer Science(UX Design) -Lecture 7

1. What Is Pseudocode (and why we use it)

Meaning: **Pseudo** = false (Greek), **Code** = instructions

- → *"Fake code" written for humans, not machines*

Purpose

- Explain **how an algorithm works**
- Capture **logic and flow** clearly
- Act as a **bridge** between:
  - Problem formulation
  - Flowcharts
  - Actual programming code

Key point: Pseudocode is written **to be understood**, not compiled.

2. Why Not Just Write Real Code?

Practical reasons

- Clients can't read real code
- Complex logic is easier to reason about in English
- Helps teams understand each other's work
- Acts as **documentation**
- Reduces bugs before coding begins
- Makes transcription to code faster

Industry reality

- Writing pseudocode first is **standard practice**
- Especially important for:
  - Large systems
  - Algorithms
  - Client-facing projects

3. Relationship Between Algorithm, Pseudocode, Flowchart

Concept	Algorithm	Pseudocode	Flowchart	Code
Purpose	List of logical steps	Human-readable version of algorithm	Visual representation of algorithm	Machine-executable version

→**Algorithm → Pseudocode → Flowchart → Code**

4. Characteristics of Good Pseudocode

Core goals

- **Clarity**, **Unambiguity**, **Readability**, **Transcribability** (Easy to turn into code)

What pseudocode is NOT

- Not tied to any programming language, Not syntactically strict, Not executable

5. Best Practices for Writing Pseudocode

1. Start with the goal

Example:

This program checks whether a number is a palindrome.

Immediately tells the reader:

- What the program does
- What to expect

2. Write in sequence

- Steps must follow logical execution order
- Especially important for complex problems

3. One statement = one action

Bad: Read number and check if it is even and display it

Good: Read number, Check if number is even & Display number

4. Use meaningful variable names

Instead of: x, y

Use: numberOne, numberTwo

- Helps both: Understanding & Transcription into code

5. Use indentation

- Shows structure
- Mirrors real programming logic
- Critical for:
  - IF blocks, LOOPS, PROCEDURES

6. Use common control keywords

Typical pseudocode keywords:

- START, END
- IF, THEN, ELSE
- WHILE, FOR, REPEAT UNTIL
- INPUT, OUTPUT
- PROCEDURE

Capitalize keywords to distinguish logic from text.

7. Avoid assumptions

- Specify:
  - Which sensor
  - Which value
  - Which condition
- No “magic happens here” steps

6. Example: Cake Baking Pseudocode (Why it works)

Why this example is strong:

- Clear procedures
- Logical sequence
- Visible decision point
- Proper indentation
- No ambiguity

It demonstrates:

- Sequential execution
- Conditional logic
- Iteration (re-bake if not done)

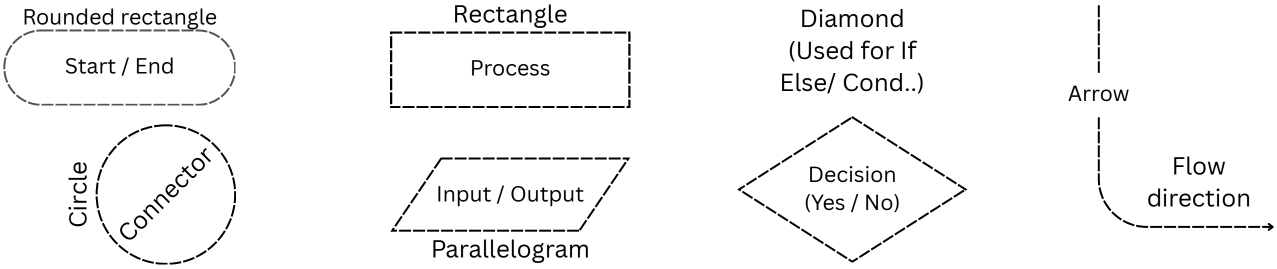
7. Flowcharts: Purpose and Role

What a flowchart does

- Visually shows:
  - Program start, Processing steps, Decisions, Loops, End state

Think of it as: A state machine diagram for a program

8. Flowchart Symbols (Must Know)



Flowcharts follow **stricter rules** than pseudocode.

9. Pseudocode vs Flowchart (Key Difference)

Aspect	Representation	Detail level	Flexibility	Rules	Best for
Pseudocode	Text	High	High	Loose	Logic
Flowchart	Visual	Medium	Lower	Strict	Process overview

They are **complementary**, not competitors.

10. Example: Even Numbers Program (What it shows)

Demonstrates:

- Input control (only 10 numbers), Looping, Decision making, Output filtering

Why it's good pseudocode

- Clear procedures, No hidden assumptions
- Easy to convert into:
  - Python, C, Java

11. Common Pseudocode Keywords (Expanded)

Keyword	INPUT	READ / GET	PRINT / DISPLAY	COMPUTE	SET / INIT	INCREMENT	DECREMENT
Meaning	Get data from user	Read from file	Output result	Perform calculation	Initialize variable	Increase value	Decrease value

12. Shape Calculation Task — How to Think

Correct approach

1. Identify requirements
2. Define valid inputs
3. Reject invalid shapes

1. Distinguish:
    - Flat shapes → area
    - Solid shapes → volume
2. Avoid repetition via procedures
3. Optimize for clarity

Key lesson: Good pseudocode improves **design quality**, not just code speed.

13. Final Takeaways

- Pseudocode is a **thinking tool**
- Flowcharts are a **visual validation tool**
- Writing code without either is:
    - Risky
    - Error-prone
    - Inefficient for complex problems
- Clear thinking → clear pseudocode → clean code

Computer Science(UX Design) -Lecture 8

1. What an Algorithm Really Is

Core idea: An **algorithm** is a **finite, step-by-step procedure** for solving a **well-specified problem**.

Key words that matter:

- **Finite** → must terminate
- **Step-by-step** → clarity and determinism
- **Well-specified** → no ambiguity

If the problem is vague, no algorithm—no matter how clever—can save you.

2. Why Studying Algorithms Matters (Even Today)

Even if:

- Computers were infinitely fast
- Memory was unlimited

We would **still** study algorithms because we must prove that a solution:

1. Exists, Is correct, Terminates, Is optimal

In the real world:

- Hardware **is limited**
- Users are **impatient**
- Efficiency **directly affects business**

88% of users abandon slow apps — optimization is not optional.

3. Algorithms vs Code (Critical Distinction)

Category	Logic	Language	Focus	Longevity
Algorithms	Focuses on logic	Language-independent	Focuses on what & how	Enduring
Code	Implementation details	Language-specific	Focuses on syntax	Changes with technology

Good developers write **code**. Great developers design **algorithms**.

4. Structure of an Algorithm

Every algorithm has **three parts**:

- 1. Input: Data the algorithm operates on
- 2. Process: Ordered steps applied to input
- 3. Output: Result produced by the process

If any of these are unclear, the algorithm is flawed.

5. Why Efficiency Is Everything

Two algorithms can: Solve the **same problem**, Produce the **same output**, But differ wildly in performance

Efficiency determines:

- Speed, Scalability, User experience, Cost

This is where **algorithm complexity** enters.

6. Algorithm Complexity (Plain English)

Algorithm complexity estimates:

How the number of steps grows as input size grows

We care about:

- **Growth rate**, not exact step count, Behavior as input becomes large

Why?

- Hardware differs, Platforms differ, Growth trends don't

7. Big-O Notation (The Language of Efficiency)

What Big-O does

- Describes **upper bound** of growth
- Ignores constants and low-order terms
- Allows fair comparison across machines

Why constants are dropped

- O(2n) and O(100n) grow the same way
- Growth rate matters more than exact counts

8. Major Time Complexities (Fast → Slow)

1. **O(1) — Constant Time**
   - Same number of steps, always

Example:

- Accessing an array element by index

Best possible performance.

2. **O(log n) — Logarithmic Time**
   - Input size is repeatedly halved

Example:

- Binary search

Extremely efficient and scalable.

3. **O(n) — Linear Time**
   - Steps grow directly with input size

Example:

- Simple loop over a list

Acceptable for moderate data sizes.

4. **O(n log n) — Linearithmic Time**
   - Common in efficient sorting algorithms

Example:

- Merge sort, quicksort (average case)

Often the **best practical balance**.

5. **O(n^2) — Quadratic Time**
   - Nested loops over input

Example:

- Bubble sort

Becomes slow very quickly as input grows.

6. **O(2^n) / O(n!) — Exponential / Factorial Time**
   - Growth explodes

Example:

- Brute-force Traveling Salesman Problem

Impractical beyond small inputs.

9. Why This Classification Matters

Imagine:

- 1,000 inputs

Complexity	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
Approx. Steps	1	~10	1,000	~10,000	1,000,000	Impossible

Same problem. Vastly different realities.

10. Algorithm Design Is a Skill

Designing algorithms means:

- Choosing the right strategy
- Balancing time vs memory
- Thinking beyond “it works”

That’s why:

- Algorithms separate engineers from coders
- Optimization expertise is highly paid
- Interview processes obsess over it

11. Final Core Takeaways

Think of algorithms as:

- **Blueprints of logic**
- **Contracts of correctness**
- **Engines of performance**

Code is just the vehicle.