Lecture 21-27 (Complete)

Concurrency and Synchronization

Introduction

- **Concurrency** means running multiple processes or threads at the same time in modern operating systems.
- It's key for **multiprogramming**, **multiprocessing**, **distributed systems**, and **client-server models**.
- Concurrency involves managing **communication**, **synchronization**, **resource sharing**, and **resource contention** among processes.
- **Process**: A program in execution with its own memory, code, data, and system resources, running independently.
- **Thread**: A lightweight unit of execution within a process, sharing its memory and resources but with its own stack.
- **Key Difference**: Processes are isolated; threads share memory, enabling faster communication but requiring synchronization.

Key Topics

- **Principles of Concurrency**: How processes run together.
- **Process Interactions**: How processes work with each other.
- **Mutual Exclusion**: Ensuring only one process accesses a shared resource at a time.
- **Solutions for Mutual Exclusion**:
  - Software: Dekker's and Peterson's algorithms.
  - Hardware: Test-and-Set operation.
  - Operating System: Semaphores.
  - Programming Language: Monitors.
  - Distributed Systems: Message passing.
- **Classic Problems**:
  - Reader/Writer Problem.
  - Dining Philosophers Problem.

Principles of Concurrency

- Concurrency involves **interleaving** (switching between processes) or **overlapping** (running processes simultaneously).
- **Example**: Two processes (P1, P2) use an "echo" function to read input and display it.
  - If P1 starts, inputs data, and gets interrupted, P2 might overwrite P1's input, causing errors.
  - This is more likely in **multiprocessing** systems with multiple processors.
- **Solution**: Control access to shared resources (like the input buffer) using **critical sections**.

Interactions Among Processes

- **Types of Interactions**:
  1. **Competing Processes**: Processes don't share data but compete for system resources (e.g., disk, printer).
     - Issues: **Deadlock** (processes stuck waiting) and **starvation** (some processes never get resources).
  2. **Cooperating Processes (Sharing):** Processes share resources like an I/O buffer.
     - Requires protecting the **critical section** to avoid conflicts.
  3. **Cooperating Processes (Communication)**: Processes coordinate without sharing data, using **synchronization**.
     - Focus is on timing and order of actions.

Race Condition

- Happens when multiple processes access and change **shared data** at the same time.
- The final result depends on which process finishes last.
- **Solution**: Synchronize processes to prevent race conditions.

Mutual Exclusion

- **Definition:** Only one process can access a **critical section** (code affecting shared resources) at a time to avoid data corruption.
- **Requirements**:

1. No assumptions about process speeds.
2. Processes stay in critical sections for a limited time.
3. Only one process in the critical section at a time.
4. Processes requesting access shouldn't wait forever.
5. Waiting processes shouldn't block others.

Software Solutions for Mutual Exclusion

- **Algorithm 1**:
    - Uses a "turn" variable to decide which process enters the critical section.
    - Problem: Forces strict alternation and causes **busy waiting** (wasting CPU time).
- **Algorithm 2**:
    - Uses a "flag" to show when a process wants the critical section.
    - Problem: Can cause **deadlock** if a process fails inside the critical section.
- **Algorithm 3 (Peterson's Algorithm)**:
    - Combines "flag" and "turn" variables.
    - Solves the critical section problem for two processes without deadlock or busy waiting.
- **Dekker's Algorithm**: Uses "two flags" and a "turn" variable to manage access to the critical section. It alternates turns between two processes and checks intent using flags.
  Solves the mutual exclusion problem without deadlock or starvation.
  Considered one of the earliest software-based synchronization solutions.

Hardware Solutions

- **Test-and-Set**:
    - Atomically checks and sets a shared Boolean variable (lock).
    - Process waits if lock is true, enters critical section if false, then resets lock.
- **Swap**:
    - Atomically swaps two Boolean variables to control access.

    - Similar to Test-and-Set but uses a different mechanism.

Semaphores

- A **semaphore** is like a counter that controls access to resources, managed by the operating system.
- **Attributes**: A value (counter) and a queue for waiting processes.
- **Operations**:
    - **wait()**: Decrements the semaphore value; if negative, the process is blocked.
    - **signal()**: Increments the semaphore value; wakes a blocked process if any.
- **Types**:
    - **Counting Semaphore**: Value can be any integer (used for multiple resources).
    - **Binary Semaphore**: Value is only 0 or 1 (like a lock).
- **Use Case**: Easily ensures mutual exclusion for any number of processes.
- **Example**:
    - Semaphore mutex starts at 1.
    - First process enters critical section (sets mutex to 0).
    - Other processes wait (decrement mutex to negative, get blocked).
    - When the process leaves, it signals mutex, waking one waiting process.

Semaphores in Action

- **Example**: Printer access.
    - Process A calls wait() to access the printer; if busy, it's blocked.
    - Scheduler runs Process B while A waits.
    - When B finishes, it calls signal(), waking A to use the printer.
- **General Synchronization**:
    - Use semaphores to ensure one process waits for another to complete a task.

    - Example: Process Pj waits for Pi to finish task A before starting task B.

Producer/Consumer Problem

- **Scenario**: A producer adds items to a shared buffer; a consumer removes them.
- **Issue**: If not synchronized, the consumer might try to take items when the buffer is empty, or the producer might overwrite data.
- **Basic Solution**:
    - Use a semaphore MUTEX (starts at 1) for mutual exclusion.
    - Producer: Adds item, updates buffer, signals MUTEX.
    - Consumer: Waits for MUTEX, takes item, signals MUTEX.
    - Problem: If the producer is slow, the consumer busy-waits (wastes CPU).
- **Improved Solution**:
    - Add semaphore AVAIL (starts at 0) to track available items.
    - Producer signals AVAIL after adding an item.
    - Consumer waits for AVAIL before taking an item, avoiding busy-waiting.
- **Bounded Buffer**:
    - Buffer has a fixed size (e.g., n slots).
    - Add semaphore BUFSIZE (starts at n) to track free slots.
    - Producer waits for BUFSIZE before adding; consumer signals BUFSIZE after taking.

Semaphores vs. Mutex

- **Mutex**:
    - A lock (object) that only the owning process can modify.
    - Either locked or unlocked; used for one resource at a time.
- **Semaphore**:
    - An integer counter modified by wait() and signal().
    - Can manage multiple resources (counting) or act like a mutex (binary).
- **Key Difference**: Mutex is for locking; semaphores are for signaling and resource counting.

Semaphore Challenges

- **wait()** and **signal()** must be atomic to avoid errors.
- Programmers might:
    - Swap wait() and signal(), causing deadlock.
    - Forget to include them, breaking mutual exclusion.
    - Scatter them across code, making it hard to manage.

Monitors

- A **monitor** is like a class that ensures only one process runs its code at a time.
- Contains:
    - Shared variables.
    - Procedures (functions) to access those variables.
    - Initialization code.
- Only one process can execute in the monitor; others wait.
- **Condition Variables**:
    - Allow processes to wait inside the monitor (e.g., x.wait() suspends the process).
    - Another process can wake it with x.signal().
- **Benefits**: Simpler to use than semaphores; reduces errors.

Semaphores vs. Monitors

- **Semaphores**: Use counters; no built-in condition variables; can be complex.
- **Monitors**: Use procedures and condition variables; easier to manage and reason about.

Message Passing

- Used for **synchronization** and **communication**, especially in distributed systems.
- **Primitives**:
    - send(destination, message): Sends a message to another process.
    - receive(source, message): Receives a message.
- **Types**:
    - **Blocking Send**: Sender waits until the message is received.

    - **Non-Blocking Send**: Sender continues without waiting.
    - **Blocking Receive**: Receiver waits for a message.
    - **Non-Blocking Receive**: Receiver continues without waiting.
- Common combo: Blocking receive with non-blocking send.

Reader/Writer Problem

- **Scenario**: Multiple processes access shared data.
    - **Readers** can read simultaneously.
    - **Writers** need exclusive access (no readers or other writers).
- **Solutions**:
    - **Readers Priority**: Readers can enter even if writers are waiting.
    - **Writers Priority**: Writers enter as soon as the critical section is free.
- **Implementation**:
    - Use semaphores ForCS (controls critical section) and ES (manages reader count).
    - Readers increment a counter (NumRdr) and wait for ForCS only if they're the first reader.
    - Writers wait for ForCS to ensure exclusive access.

Dining Philosophers Problem

- **Scenario**: Five philosophers sit around a table, each needing two forks to eat.
    - Forks are shared between neighbors.
    - Goal: Avoid deadlock (everyone grabs one fork) and starvation.
- **Solution Using Monitors**:
    - Monitor dp tracks each philosopher's state (thinking, hungry, eating).
    - **pickup(i)**: Philosopher i becomes hungry, checks if forks are free, waits if not.
    - **putdown(i)**: Philosopher i finishes eating, signals neighbors to check forks.
    - **test(i)**: Checks if philosopher i can eat (neighbors not eating) and signals them.

    - Initializes all philosophers as thinking.

Summary

- Synchronization is critical for concurrent processes to avoid conflicts.
- Solutions range from:
    - **Hardware**: Test-and-Set, Swap.
    - **Software**: Dekker's, Peterson's algorithms.
    - **OS**: Semaphores.
    - **Programming**: Monitors.
    - **Distributed Systems**: Message passing.
- Kernel-level synchronization often uses hardware locks or non-preemptive kernels.

Code Tuning

Definition

Code tuning is the process of **modifying already correct code** to improve its **efficiency**, such as making it run **faster**, use **less memory**, or perform **fewer operations**.

⚠ Note: While tuning improves performance, it can **reduce code reliability** and **readability** if done improperly.

Focus Areas of Code Efficiency

Efficiency is not only about optimizing lines of code—it involves:

1. **Program Design** – Choosing efficient architecture and flow.
2. **Module & Routine Design** – Organizing small units of code to reduce overhead.
3. **Operating System Interactions** – Managing how code communicates with OS features like file I/O.
4. **Code Compilation** – Taking advantage of compiler optimizations.

5. **Hardware Utilization** – Writing code that works well with the system's CPU, memory, and cache.

Elements of Code Tuning

1. **Measure Before You Modify**
   - Don't guess—**use profiling tools** to find performance bottlenecks.
   - Example: Tools like VisualVM, Perf, or Python's cProfile.
2. **Identify Inefficient Areas (The 80/20 Rule)**
   - Often, **80% of execution time** is spent in **20% of the code**.
3. **Repeat Optimization (Iteration)**
   - Optimization is **not one-time**—keep testing and improving in cycles.
4. **Common Bottlenecks:**
   - **Input/Output Operations**: Slow file reading/writing.
   - **Paging**: Excessive memory access leads to slowdowns.
   - **System Calls**: Repeated OS interactions c an be costly.

;

Real-Life Example

Example: A Web App Loading Slowly
A developer notices their website takes too long to load. Instead of rewriting everything:

- They use **profiling tools** to check what takes time.
- It turns out image files are too large and JavaScript files aren't minified.
- Instead of changing the app logic, they **optimize the assets** (e.g., compressing images, minifying JS).
- Result: The site loads **2x faster** without touching most of the core code.

Construction Tools

Definition

Construction tools are software applications or utilities used during software development to **increase productivity** and **enhance software quality**.

These tools are divided into four broad categories:

1. **Source Code Tools**
2. **Design Tools**
3. **Code Quality Analyzation Tools**
4. **Executable-Code Tools**

---

1. Source Code Tools

These tools help with **editing**, **managing**, and **understanding source code** more efficiently.

Tool Type	**Function**
Editors	Basic tools to add, delete, or rearrange code. *(e.g., VS Code, Sublime Text)*
File Comparators	Compare two files and highlight differences. *(e.g., WinMerge, diff)*
Source-Code Beautifiers	Format code uniformly to enhance readability. *(e.g., Prettier, ClangFormat)*
Templates	Reusable code blocks to save time and maintain consistency.
Browsers	Allow searching and navigating code. *(e.g., GitHub code browser)*

Tool Type	**Function**
Cross-Reference Tools	Show where variables or functions are defined and used. *(e.g., ctags, LXR)*

---

2. Design Tools

Design tools assist in creating and organizing **software design visually**, which is especially helpful for planning system architecture.

Tool Type	**Purpose**
Graphic Drawing Tools	Help in visual modelling of the system architecture.
Support Object-Oriented and Structured Design	Examples: UML diagrams for OOD, Data Flow Diagrams for structured design.
Entity-Relationship Diagrams (ERD)	Show relationships between data entities in a system.
CASE Tools (Computer-Aided Software Engineering)	Manage design consistency, update diagrams automatically, and track system changes. *(e.g., Enterprise Architect, StarUML)*

🧠 **Real-Life Example:** When a new module is added to a software project, CASE tools automatically update all related diagrams to maintain design consistency.

---

3. Code Quality analyzation Tools

These tools help in **examining code for structure, readability, maintainability, and performance.**

Tool Type	**Function**
Call-Structure Generators	Display which routines call which other routines.
Syntax & Semantic Checkers	Catch deeper errors missed by compilers. *(e.g., Lint tools)*

Tool Type	Function
Metrics Reporters	Report complexity, size, and quality of code. *(e.g., Cyclomatic complexity, LOC)*
Restructurers	Clean up messy ("spaghetti") code into organized structure.
Code Translators	Convert code from one language to another. *(e.g., Java to Kotlin converter)*
Data Dictionary	A database that stores variable and function names with descriptions for better understanding and traceability.

---

4. Executable-Code Tools

These tools work with **compiled or executable code** to support building, testing, debugging, and optimizing.

Tool Type	Purpose
Linkers	Combine object files into a single executable.
Code Libraries	Prebuilt modules that can be reused in programs. *(e.g., jQuery, .NET libraries)*
Code Generators	Create code from models or templates. Useful for prototyping. *(e.g., low-code platforms)*
Macro Preprocessors	Define constants or macros that simplify code without runtime cost. *(e.g., #define in C)*
Debuggers	Step through code to find and fix bugs. *(e.g., GDB, Chrome DevTools)*
Execution Profilers	Analyze code during runtime to find performance bottlenecks. *(e.g., Perf, Valgrind, JProfiler)*
Assembler Listings	Convert human-readable assembler to machine code for execution.

🧠 **Real-Life Example:** A game developer uses a profiler to discover that a rendering loop is taking 70% of CPU time. After optimizing it, the game runs 30% smoother.

Design Thinking – Six Stages Explained

Design Thinking is a **human-centered**, **iterative approach** to **problem-solving** that focuses on **understanding users**, **challenging assumptions**, and **developing innovative solutions**.

Why Use Design Thinking?

- Helps solve **complex problems** in creative ways.
- Focuses on the **user's real needs**, not just technical requirements.
- Encourages **team collaboration** and **testing ideas quickly**.

🧠 **Real-life Example:** Think of how Apple designs its products. They focus on how people use them, not just how they are built. That's Design Thinking.

Here are the **six stages** of Design Thinking:

---

1. Empathize

- **Purpose:** Understand the user's needs, experiences, and emotions.
- **Activities:** Observe users, conduct interviews, and engage with them to gain deep insight into their challenges.
- **Example:** A team designing a fitness app interviews people who struggle to maintain workout routines to understand their motivations and obstacles.

---

2. Define

- **Purpose:** Clearly state the problem based on user needs discovered in the Empathize stage.

- **Activities:** Organize information, identify patterns, and write a problem statement (also called a Point of View).
- **Example:** "Young professionals need a way to exercise efficiently at home because they lack time and access to gyms."

---

3. Ideate

- **Purpose:** Generate a wide range of creative ideas to solve the problem.
- **Activities:** Brainstorming, mind mapping, and sketching potential solutions without judging them.
- **Example:** The team proposes ideas like a 10-minute workout video app, virtual fitness coach, or AI-generated workout plans.

---

4. Prototype

- **Purpose:** Build simple versions of one or more ideas to explore potential solutions.
- **Activities:** Create low-cost, basic models (digital or physical) that allow testing the idea.
- **Example:** Develop a rough mobile app mock-up or paper interface that shows how the workout app would function.

---

5. Test

- **Purpose:** Evaluate the prototypes with real users to gather feedback.
- **Activities:** Let users interact with the prototype and observe their behaviour. Use their responses to refine the solution.
- **Example:** Users test the app prototype and mention that the interface is confusing, so the team simplifies it.

---

6. Implement

- **Purpose:** Launch the final version of the solution after refining it based on feedback.

- **Activities:** Final development, real-world deployment, and continuous improvement.
- **Example:** The final version of the app is released on app stores, and the team continues to monitor user reviews for further improvements.

Lecture 17-20 (Complete)

1. User Interface (UI) Design – What & Why?

What is UI Design?

UI (User Interface) Design is how a product **looks and works on the screen**. It includes all the visual elements like **buttons, colours, icons, fonts, spacing, and layout**.

Why is UI important?

Good UI helps users **understand and use** the app easily without confusion.

Example:

Instagram uses a **simple layout** with clean icons and easy navigation. The buttons for "Home," "Search," "Post," "Reels," and "Profile" are always in the same place, so users quickly know where to go.

---

2. User Experience (UX) Design – What & Why?

What is UX Design?

UX (User Experience) Design is about **how a user feels** when using a product. It focuses on making the experience **smooth, fast, and satisfying**.

Why is UX important?

A good UX help users **complete tasks easily**, keeps them happy, and **makes them want to use the app again**.

Example:

In Amazon’s app, users can **search, filter, buy, and track products** very smoothly. That’s good UX—simple steps, clear info, and fast loading.

---

3. UI vs UX (What’s the difference?)

UI Design	**UX Design**
How the product **looks**	How the product **feels and works**
Focuses on **colours, layout, style**	Focuses on **user journey and experience**
Layout, Visual Designing, Branding	Usability Testing, User Research, Personas
Example: Button design	Example: What happens when user clicks the button

User Persona: A User Persona is a fictional character created to represent a typical user of a product or service. It’s based on real user data and helps teams understand users’ needs, goals, behaviour, and challenges so they can design better experiences.

Simple Example:
In the Facebook app:

- **UI** is the blue color, icons, and fonts.
- **UX** is how easy it is to scroll, comment, or like a post.

---

4. The Design Process (5 Steps)

Based on **Jesse James Garrett's** UX process.

1. Strategy

- Understand **why** we are building the product and **who** the users are.
- What are the **goals** of the users and the business?

Example: Instagram's goal = Share moments with friends.
User goal = Post photos easily.

---

2. Scope

- Decide what **features** and **functions** will be included.
- Define the content (texts, images, etc.) and what users can do.

Example: In TikTok:
Scope includes video upload, likes, comments, and sharing.

---

3. Structure

- Organize **how users will move** through the product.
- Create the **navigation flow and logic**.

Example: In YouTube:
Users can search → click a video → scroll to comments or next videos.

---

4. Skeleton

- Build a **basic wireframe or layout** to show where things will go.
- Focus on **placement** of buttons, text, images, etc.

Example: A wireframe of a login screen showing where to type email and password.

---

5. Surface

- Add the final **visual design**—colours, fonts, icons, images.
- This is what the user sees and interacts with.

Example: WhatsApp's green color theme, message bubbles, icons.

---

5. UI Design Core Principles (With Examples)

1. Clarity

Everything should be **easy to see and understand**.

Example: Google's homepage—just a search bar. No confusion.

---

2. Feedback

The system should **respond to user actions,** so they know it's working.

Example: In Instagram, when you like a post, the heart icon turns red.

---

3. Consistency

Keep the **design style same** across all screens.

Example: Facebook uses the same blue theme, fonts, and buttons on all pages.

---

4. Visual Hierarchy

Design elements should show what's **most important**.

Example:

- **Typography:** Bigger text for headings, smaller for details.
- **Spacing:** WhatsApp uses space between chats to make it readable.
- **Colours:** Red for errors, green for success.

---

5. Established Design Patterns

Use **familiar design styles** so users don't get confused.

Examples:

- **Lazy Registration Pattern:** Let users try the app before signing up (like Canva lets you design before asking to sign in).
- **Google Autocomplete:** Suggests search terms to save time.

- **Hamburger Menu:** Used in many apps like YouTube and LinkedIn for hidden menus.

Code Refactoring and Review

What is Code Refactoring?

Simple Definition:

Refactoring means **cleaning and improving your code** without changing what it does. It's like **cleaning your room**—you don't remove anything, but you make it neat and easy to find things.

Why Refactor Code?

- Makes code **easier to understand**
- Makes code **shorter and cleaner**
- Helps **remove bugs or errors** in the future
- Makes adding new features **faster**

- Improves **performance and speed**

Tools for Refactoring:

Tool	**Use**
Visual Studio Code	Has built-in tools like **"Rename symbol"**, **"Extract method"**
PyCharm / IntelliJ	Smart suggestions to **clean up code** automatically
Eslint	Helps clean up **JavaScript code** and follows best practices
Prettier	Formats code automatically (clean and consistent style)

Examples of Refactoring:

🧱 Before: Repeating Code

```
print("Hello, Ali")

print("Hello, Sara")

print("Hello, Umar")
```

✨ After: Using a Function

```
def greet(name):

   print(f"Hello, {name}")


greet("Ali")

greet("Sara")

greet("Umar")
```

What is Code Review?

Definition:

Code Review means that **another developer checks your code** before it's finalized. It helps catch mistakes, share knowledge, and write better quality code.

Why Code Reviews are Important?

- **Catch bugs early**
- Learn from other developers
- Make sure code is **easy to understand**
- Maintain **coding standards**
- Encourage **teamwork**

🔧 **Tools for Code Review:**

Tool	**Use**
GitHub	Review code using **pull requests** and leave comments
GitLab	Similar to GitHub, great for teams
Bitbucket	For code sharing and reviews with inline comments
Phabricator	Strong review workflow for larger teams

Example of Code Review Process:

1. Developer writes code and **pushes to GitHub**
2. Opens a **Pull Request** (PR)
3. Another team member checks it:
   - Is the logic correct?
   - Is the code readable?
   - Are variable names clear?
   - Are there any security risks?
4. They leave **comments or suggestions**
5. Developer makes changes and **merges the code**

🎯 User Persona – Thele Wala (Fruit & Vegetable Seller in Pakistan)

🧑‍💼 **Name: Rashid Bhai**

👤 **Age: 42 years**

🏠 **Lives in: Lahore, Pakistan**

🛒 **Job: Pushcart Fruit & Vegetable Seller (Thela wala)**

📅 **Experience: 15 years selling in local streets and markets**

🧠 Goals:

- Accurately **weigh fruits and vegetables** for customers
- **Track daily sales** easily without pen and paper
- **Save money** safely, and know how much profit he makes
- **Look professional** to attract more customers
- Spend **less time doing math**—more time selling

😟 Pain Points (Problems):

- Old manual scales are **not accurate**, and sometimes customers complain
- He **forgets how much he earned** or spent in a day
- Hard to **keep track of which items sell more**
- Doesn't know how to **save money properly**
- Doesn't understand complex mobile apps or digital systems

📱 Technology Use:

- Owns a **basic Android phone**
- Knows how to use **WhatsApp and YouTube**
- Can read **simple Urdu or Roman Urdu**

🧩 Needs from the Machine:

- **Digital weight scale** that is easy to use
- A **simple screen or voice feature** that tells how much money to take
- **Auto-record of daily sales**—in numbers or even voice notes
- A way to **see which item sold the most** in one day or week
- Option to **see savings or profits** clearly
- Machine should be **low-cost, battery-powered**, and **durable**

🛠 Design Ideas (based on persona needs):

- Use **Urdu voice output** (e.g., "Aapka bill Rs. 250")
- Show data with **big icons or graphs** (for items sold)
- Auto-save sale data with **one-button press**
- Rechargeable battery or solar-powered
- Daily summary like: “Aaj aapne 3 kilo tamatar aur 5 kilo seb beche. Total Rs. 1200.”

🔁 Recursion and Iteration – In-depth Notes (Theory Only)

🔷 What is Recursion?

Recursion is a method where a function **calls itself** to solve smaller instances of a problem until it reaches a base case (a condition that stops the recursion). It **breaks down** a problem into smaller subproblems of the same type.

🔷 What is Iteration?

Iteration is a technique where a set of instructions is **repeated** using loops (like "for", "while", or "do-while") until a specific condition is met. It uses **repetition instead of function calls**.

🔷 Why Use Recursion or Iteration?

Goal	**Use Recursion When**	**Use Iteration When**
Simplifying complex logic	Problem can be broken into smaller parts	Problem can be solved with a repeated process
Code readability	Helps express divide-and-conquer problems	Better when logic is simple and repetitive
Memory efficiency	Less preferred (can use more stack memory)	Preferred (uses constant memory)

🔷 How Do They Work?

- **Recursion**:
  - Solves a problem by dividing it into smaller, similar subproblems.
  - Each recursive call adds a new frame to the call stack.
  - Stops when it reaches a base case.
- **Iteration**:
  - Repeats a block of statements.
  - Uses looping control variables.
  - Stops when a loop condition fails.

🔷 Where Are They Used?

Recursion is often used in:

- Tree/graph traversals
- Backtracking (e.g., solving a maze, Sudoku)
- Divide-and-conquer algorithms (e.g., Merge Sort, Quick Sort)
- Mathematical computations (like factorial, Fibonacci)
- File system navigation

Iteration is used in:

- Counting and summing
- Repeated input/output
- Looping over arrays, lists, and other data structures
- Basic algorithm implementations (e.g., linear search)

🔷 Which is Better?

- **Iteration is better** when performance and memory usage matter.
- **Recursion is better** when the problem has a **recursive nature** and needs clean, short code (e.g., tree operations).
- In many cases, recursion can be **converted to iteration** for better performance.

🔷 Types of Recursions

1. **Direct Recursion**: Function calls itself directly.
2. **Indirect Recursion**: Function A calls B, and B calls A.
3. **Tail Recursion**: Recursive call is the last thing in the function.
4. **Head Recursion**: Recursive call occurs before other operations.
5. **Tree Recursion**: Multiple recursive calls per function call.

🔷 Types of Iteration

1. **Entry-Controlled**:
   - Condition checked before executing (e.g., while, for)
2. **Exit-Controlled**:
   - Loop executes at least once (e.g., do-while)

🔷 Famous Problems & Their Typical Solutions

Problem	**Preferred Method**	**Reason**
Factorial	Recursion/Iteration	Naturally recursive but can be easily iterated
Fibonacci Sequence	Iteration (for large n)	Iteration is faster and avoids stack overflow
Binary Search	Both	Clean in recursion; more efficient in iteration
Tree Traversal (DFS, etc.)	Recursion	Matches the recursive structure of trees
Tower of Hanoi	Recursion	Each move depends on previous subproblems
Sorting (Merge/Quick Sort)	Recursion	Divide-and-conquer logic
Looping over arrays	Iteration	Straightforward, efficient

🔷 Advantages and Disadvantages

✅ Advantages of Recursion

- Clean and concise code.
- Easier to implement for naturally recursive problems.
- Matches divide-and-conquer approach.
- Useful for navigating hierarchical structures.

❌ Disadvantages of Recursion

- Can cause **stack overflow** if not controlled.
- Slower due to overhead of function calls.
- More memory usage (call stack).

✅ Advantages of Iteration

- **Memory-efficient** (no call stack usage).
- **Faster execution** in most cases.
- Better for performance-critical applications.

❌ Disadvantages of Iteration

- Code can be more **complex** and harder to read in some cases.
- Doesn’t work well for problems that are naturally recursive (e.g., tree traversal).

🔷 Tabular Comparison: Recursion vs Iteration

Feature	**Recursion**	**Iteration**
Definition	Function calls itself	Repeats block using loops
Control Structure	Function call	Loops (for, while, etc.)

Feature	Recursion	Iteration
Termination	Base case	Condition failure
Memory Use	More (uses call stack)	Less (uses fixed variables)
Speed	Slower (overhead of calls)	Faster
Code Clarity	Cleaner for complex logic	Clear for simple repetitive logic
Risk of Errors	Stack overflow, infinite recursion	Infinite loop
When to Use	Problems with recursive nature	Problems with repeated computation
Examples	Tree traversals, Tower of Hanoi	Searching arrays, summing numbers
Complexity Handling	Better for divide-and-conquer logic	Better for basic tasks

---

Code Refactoring and Code Review – Theoretical Notes

What is Code Refactoring?

Code Refactoring is the process of **restructuring existing code** without changing its external behaviour. The goal is to **improve internal structure**, readability, maintainability, and performance.

Think of it like "cleaning up" code—removing redundancies, simplifying logic, and organizing functions better.

What is Code Review?

Code Review is the process where **developers inspect each other's code** to catch mistakes, ensure consistency, and improve code quality. It's a collaborative practice, often done **before code is merged** into the main project.

🔷 Why is Code Refactoring Important?

- Improves **code readability** and understandability
- Makes future **maintenance easier**
- Reduces **technical debt**
- Enhances **performance** and scalability
- Encourages **modular and reusable** code design

🔷 Why is Code Review Important?

- Identifies **bugs** early in the development cycle
- Enforces **coding standards and best practices**
- Facilitates **knowledge sharing** among team members
- Improves **security** and **robustness**
- Enhances **team collaboration** and feedback culture

🔷 How is Code Refactoring Done?

- Performed incrementally, often during feature additions or bug fixes
- Usually includes:
  - Renaming variables/methods for clarity
  - Breaking large functions into smaller ones
  - Removing duplicate code
  - Reorganizing code blocks or modules
- Supported by automated tools and IDEs

🔷 How is Code Review Conducted?

- Typically done in one of the following ways:
    - **Pair Programming**: Real-time review while coding
    - **Over-the-Shoulder**: Developer explains the code to a reviewer
    - **Tool-based Reviews** (e.g., GitHub, GitLab): Asynchronous commenting and suggestions
- Reviewers look for:
    - Logic errors
    - Coding standard violations
    - Poor design or unclear structure
    - Security vulnerabilities
    - Documentation and comments

🔷 When and Where Are They Used?

Task	**When it is Used**	**Where it Applies**
Code Refactoring	During or after feature addition or bug fixes	Within IDEs, development environments
Code Review	Before merging code to main branches (PR stage)	Within version control platforms (e.g., GitHub, Bitbucket)

🔷 Benefits of Code Refactoring

- Cleaner, shorter code
- Easier onboarding for new developers

- Helps avoid “spaghetti code”
- Enhances system performance (in some cases)

🔷 Challenges of Code Refactoring

- Might introduce bugs if not tested well
- Can take time with little visible output
- Needs strong understanding of system behaviour

🔷 Benefits of Code Review

- Early detection of bugs
- Improves team code consistency
- Encourages accountability
- Educates junior developers through feedback

🔷 Challenges of Code Review

- Time-consuming if not scoped properly
- Can lead to friction if not done respectfully
- Depends on reviewer expertise and context

🔷 Tabular Comparison: Code Refactoring vs Code Review

Feature	**Code Refactoring**	**Code Review**
Definition	Improving internal code without changing behaviour	Peer-reviewing code for quality and correctness
Goal	Enhance structure, readability, and maintainability	Catch bugs, enforce standards, and ensure quality

Feature	**Code Refactoring**	**Code Review**
Performed By	Developer who wrote the code	Other developers or peers
When It Happens	During development or maintenance	Before merging code
Output	Cleaner, more efficient code	Reviewed and approved code
Tools Involved	IDEs, linters, static analysis tools	GitHub, Bitbucket, GitLab (code review tools)
Risk	Might unintentionally break functionality	May slow down delivery if overdone
Skill Required	Deep understanding of the codebase	Critical thinking, communication, code knowledge