

Oposiciones cuerpo de secundaria.

Esquemas sobre temario oposición profesorado Secundaria.

Especialidad informática

Autor: Sergi García Barea

Actualizado Mayo 2025



Reconocimiento – NoComercial – CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Índice

Introducción	3
Para el buen docente	3
¿Para qué prueba están adaptados estos esquemas?	3
Tema 1: Representación y comunicación de la información	4
Tema 2: Elementos funcionales de un ordenador digital. Arquitectura	8
Tema 3: Componentes, estructura y funcionamiento de la Unidad Central de Proceso (CPU)	13
Tema 4: Memoria Interna: Tipos, Direccionamiento, Características y Funciones	16
Tema 10: Representación Interna de los Datos	20
Tema 11: Organización Lógica de los Datos. Estructuras Estáticas	24
Tema 12: Organización Lógica de los Datos – Estructuras Dinámicas	28
Tema 13: Ficheros: Tipos, Características, Organizaciones	32
Tema 16: Sistemas Operativos: Gestión de Procesos	36
Tema 20: Explotación y administración de sistemas operativos monousuario y multiusuario	47
Tema 21: Sistemas informáticos. Estructura física y funcional	51
Tema 22: Planificación y explotación de sistemas informáticos. Configuración. Condiciones de instalación. Medidas de seguridad. Procedimientos de uso.	55
Tema 23: Diseño de algoritmos. Técnicas descriptivas.	59
Tema 24: Lenguajes de programación: Tipos y características	64
Tema 25: Programación Estructurada. Estructuras Básicas. Funciones y Procedimientos.	69
Tema 27: Programación orientada a objetos. Objetos. Clases. Herencia. Polimorfismo. Lenguajes.	79
Tema 31: Lenguaje C: Características generales. Elementos del lenguaje. Estructura de un programa. Funciones de librería y usuario. Entorno de compilación. Herramientas para la elaboración y depuración de programas en lenguaje C.	84
Tema 32: Lenguaje C. Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.	91
Tema 36: La manipulación de datos. Operaciones. Lenguajes. Optimización de consultas.	97
Tema 39: Lenguajes para la definición y manipulación de datos en sistemas de Bases de Datos Relacionales. Tipos. Características. Lenguaje SQL	102
Tema 44. Técnicas y Procedimientos para la Seguridad de los Datos	107
Tema 54: Diseño de Interfaces Gráficas de Usuario (GUI)	111
Tema 60: Sistemas basados en el conocimiento. Representación del conocimiento. Componentes y arquitectura.	114
Tema 61: Redes y Servicios de Comunicaciones	118
Tema 62: Arquitecturas de sistemas de comunicaciones: capas y estándares	121
Tema 72: Seguridad en Sistemas en Red: Servicios, Protecciones y Estándares Avanzados	125
Tema 74: Sistemas Multimedia	129
Guía práctica de estrategias docentes para Informática (Apoyo)	132

Introducción

Este documento recoge una serie de **esquemas sintéticos del temario oficial para las oposiciones al cuerpo de profesorado de Secundaria, especialidad Informática**, con el objetivo de ofrecer una herramienta de estudio clara, útil y eficaz. Cada esquema está diseñado para ocupar como máximo **cuatro páginas**, facilitando así su consulta rápida, comprensión global y memorización eficaz.

Para el buen docente

Pero estos esquemas **no son solo para superar una oposición**. Están pensados para ayudarnos a **ser mejores docentes**, personas que entienden la complejidad técnica de su materia, pero también su dimensión educativa, social y ética. Ser docente es una tarea de gran responsabilidad que trasciende un examen: **enseñamos a través de lo que sabemos, pero también a través de lo que somos**.

Por eso, si has llegado hasta aquí, te pido algo importante: lleva contigo el compromiso de ser un buen docente más allá de la oposición. Utiliza estos materiales como base, sí, pero hazlos crecer con tu experiencia, tus reflexiones y tu vocación. Que enseñar sea una decisión consciente, diaria, y no un trámite. Que lo que prepares hoy, lo apliques con compromiso durante toda tu carrera docente, pensando siempre en lo mejor para tu alumnado.

¿Para qué prueba están adaptados estos esquemas?

Estos esquemas están específicamente adaptados para la **prueba de exposición oral del procedimiento selectivo regulado por la ORDEN 1/2025, de 28 de enero**, de la Conselleria de Educación, Cultura, Universidades y Empleo de la Comunitat Valenciana, que establece lo siguiente:

"La exposición tendrá dos partes: la primera versará sobre los aspectos científicos del tema; en la segunda se deberá hacer referencia a la relación del tema con el currículum oficial actualmente vigente en el presente curso escolar en la Comunitat Valenciana, y desarrollará un aspecto didáctico de este aplicado a un determinado nivel previamente establecido por la persona aspirante. Finalizada la exposición, el tribunal podrá realizar un debate con la persona candidata sobre el contenido de su intervención."

No obstante, estos materiales pueden ser también útiles para preparar **otras modalidades de oposición** (como ingreso por estabilización o pruebas de adquisición de especialidades), así como para otras especialidades cercanas, especialmente **la de Sistemas y Aplicaciones Informáticas**, ya que comparten gran parte del temario técnico

Tema 1: Representación y comunicación de la información

1. Introducción General

- ¿Qué es la representación digital?
- Importancia: eficiencia, seguridad, interoperabilidad.
- Ámbitos de aplicación: programación, redes, almacenamiento, computación en la nube.

2. Sistemas de Numeración

- Conceptos: base, dígitos, sistema posicional.
- Sistemas principales:
 - Decimal (base 10)
 - Binario (base 2)
 - Octal (base 8)
 - Hexadecimal (base 16)
- Utilidad en arquitectura de computadoras y codificación en redes.

3. Representación de la Información

3.1 Datos numéricos

- Enteros con y sin signo (CA1, CA2, Exceso K). Se usa CA2 por simplificar circuitería.
- Punto flotante: IEEE 754 (precisión simple y doble)
 - Simple: 1 bit para el signo (s) del número, 23 bits para la mantisa (m) y 8 bits para el exponente (exp)
 - Doble: 1 bit para el signo (s) del número, 52 bits para la mantisa (m) y 11 bits para el exponente (exp).

3.2 Texto y símbolos

- ASCII, Unicode (UTF-8, UTF-16, UTF-32)
- Aplicación en interfaces, APIs REST, microservicios

3.3 Otros tipos de datos

- Imágenes, audio, vídeo (representación binaria)
- Compresión y codificación en la nube (JPEG, MP3, MP4, AV1)

4. Conversiones entre Bases

- Binario ↔ Decimal
- Binario ↔ Hexadecimal / Octal

- Aplicaciones:
 - Debugging
 - Ensamblador
 - Dirección de memoria
 - Desarrollo de software y redes

5. Operaciones Binarias y Lógica Digital

- Aritmética: suma, resta, multiplicación, división
- Números negativos: CA1 y CA2
- Lógica: AND, OR, XOR, NOT
- Aplicación en hardware, electrónica digital y programación a bajo nivel

6. Códigos Binarios

6.1 Numéricos

- BCD, Exceso-3, Código Gray
- Aplicación en sensores digitales y sistemas embebidos

6.2 Alfanuméricos

- ASCII, Unicode
- Soporte internacional en sistemas distribuidos

7. Detección y Corrección de Errores

7. Detección y Corrección de Errores

Durante la transmisión o almacenamiento de datos pueden producirse errores por ruido, fallos eléctricos o defectos en los medios. Existen mecanismos que permiten **detectar** si un dato ha sido alterado, e incluso **corregirlo automáticamente**.

Bit de Paridad

- Añade un bit extra que indica si el número total de 1s debe ser **par o impar**.
- Solo detecta errores de **un único bit**, pero **no los corrige**.
- Usado en comunicaciones simples (por ejemplo, puertos serie antiguos).

CRC (Cyclic Redundancy Check)

- Aplica una operación matemática sobre el mensaje y transmite un código de verificación.
- El receptor recalcula este código para comprobar que los datos están íntegros.
- Detecta muchos tipos de errores, incluidos **errores en ráfaga**.
- Muy usado en **Ethernet**, discos duros, y formatos como ZIP o PNG.

Código de Hamming

- Inserta múltiples bits de paridad en posiciones estratégicas.
- Permite **corregir errores de un solo bit** y detectar errores dobles.
- Se aplica en **memorias ECC**, donde la fiabilidad es crítica (servidores, sistemas embarcados).

Reed-Solomon

- Diseñado para trabajar sobre bloques de datos, corrigiendo **múltiples errores consecutivos**.
- Ideal cuando hay muchas interferencias o pérdidas.
- Se utiliza en **RAID**, **CD/DVD**, **códigos QR**, y en **comunicaciones satelitales**.

8. Representación en la Nube y Big Data

- **JSON / BSON**: formatos estructurados; JSON es legible, BSON es binario y más rápido (usado en MongoDB).
- **Avro / Protobuf**: binarios compactos, ideales para transmitir grandes volúmenes entre servicios (e.g., Kafka, microservicios).
- **Parquet / ORC**: almacenan datos por columnas, lo que reduce tamaño y acelera análisis (usados en Spark, AWS Athena).
- **APIs cloud-native**: usan formatos como JSON o Protobuf para facilitar comunicación entre servicios distribuidos (REST, gRPC).
- **S3, Azure Blob, HDFS**: sistemas de almacenamiento escalables que soportan estos formatos y permiten análisis masivo desde la nube.

9. Seguridad y Protección de la Información

9.1 Hashing

- Propiedades: unidireccionalidad, integridad
- Algoritmos: SHA-256, bcrypt, Argon2
- Aplicaciones: autenticación, blockchain, contraseñas seguras

9.2 Cifrado

- Simétrico: AES
- Asimétrico: RSA, ECC
- Aplicaciones: HTTPS, VPN, cifrado en la nube (BitLocker, TLS, AWS KMS)

10. Comunicación Digital

10.1 Modelo de Comunicación

- Shannon-Weaver: emisor, canal, receptor, ruido
- Señales analógicas vs digitales
- Medios: cobre, fibra, Wi-Fi, 5G

10.2 Protocolos y Estándares

- TCP/IP, Ethernet, HTTP/2, QUIC
- Aplicaciones modernas: Zoom, WebRTC, videollamadas seguras

11. Compresión de Datos

- **Sin pérdida:** Huffman, LZ77, LZW
- **Con pérdida:** JPEG, MP3, H.264, AV1
- Uso en streaming, backups, edge computing y almacenamiento cloud

12. Conclusión

- La representación de datos es la base del funcionamiento digital.
- Su dominio permite optimizar rendimiento, seguridad y escalabilidad.
- Esencial para comprender desde microcontroladores hasta arquitecturas cloud-native.

Actividad: “Del bit al mensaje: simulando la vida de los datos”

Idea de la actividad:

El alumnado participará en una dinámica práctica en la que simulará el recorrido completo de un dato digital, desde su representación binaria hasta su transmisión segura. Organizados en equipos, cada grupo representará una etapa del proceso:

1. Conversión entre bases numéricas: codificar una cantidad desde decimal a binario, octal y hexadecimal.
2. Codificación de caracteres: transformar una palabra en binario usando ASCII o Unicode.
3. Detección de errores: aplicar bit de paridad o código de Hamming a un mensaje binario simulado.
4. Verificación de integridad: calcular un hash simple para comprobar que el mensaje no ha sido modificado.
5. Transmisión: representar cómo se enviaría ese mensaje a través de un canal (analógico o digital, simulado en clase).

Cada grupo resolverá su parte y pasará el “mensaje” al siguiente, que trabajará sobre la salida anterior. Al final, se verificará si el mensaje recibido coincide con el original. Esta actividad permite vivenciar de forma secuencial y colaborativa los procesos clave de representación, codificación y transmisión de datos en informática.

Tema 2: Elementos funcionales de un ordenador digital. Arquitectura

1. Introducción

Un ordenador digital es un sistema capaz de procesar datos automáticamente mediante instrucciones programadas. Su comportamiento y rendimiento dependen de su **arquitectura**, es decir, la forma en que se organizan y conectan sus componentes funcionales. Aunque los elementos básicos son comunes, su disposición varía según el modelo arquitectónico.

2. Elementos funcionales de un ordenador

Todo sistema informático moderno se compone de:

- **CPU (Unidad Central de Proceso):** ejecuta instrucciones, opera con datos, y toma decisiones.
- **Memoria principal:** almacena temporalmente datos e instrucciones que necesita la CPU.
- **Unidad de Entrada/Salida (E/S):** conecta el sistema con el entorno (dispositivos, red, usuario).
- **Sistema de buses:** interconecta todos los componentes y permite el flujo de datos, direcciones y señales de control.

Estos elementos conforman el núcleo de cualquier arquitectura computacional (Von Neumann, Harvard, etc.).

3. Modelos arquitectónicos

3.1 Arquitectura Von Neumann

- Memoria única para datos e instrucciones.
- Ejecución secuencial.
- Problema: “cuello de botella” entre CPU y memoria.

3.2 Arquitectura Harvard

- Memoria separada para instrucciones y datos.
- Acceso simultáneo a ambas memorias → mayor eficiencia.
- Usado en sistemas embebidos y microcontroladores.

4. Taxonomía de Flynn

Clasifica las arquitecturas según el número de instrucciones y datos que pueden procesar simultáneamente:

- **SISD**: una instrucción, un dato. Arquitectura secuencial tradicional (ej. Intel 8086).
- **SIMD**: una instrucción, múltiples datos. Usado en GPUs o instrucciones vectoriales (SSE/AVX).
- **MISD**: múltiples instrucciones, un dato. Arquitectura teórica, usada en sistemas críticos.
- **MIMD**: múltiples instrucciones, múltiples datos. Base de los sistemas multicore modernos.

5. Unidad Central de Proceso (CPU)

5.1 Registros

- Almacenamiento interno ultrarrápido.
- Tipos: de propósito general, de control (PC, IR, FLAGS), de memoria (MAR, MDR).

5.2 ALU (Unidad Aritmético-Lógica)

- Ejecuta operaciones básicas y lógicas.
- Soporta instrucciones SIMD y operaciones en coma flotante (FPU).

5.3 Unidad de Control

- Coordina la ejecución de instrucciones mediante señales.
- Tipos: cableada (rápida), microprogramada (flexible).

6. Jerarquía y tipos de memoria

6.1 Jerarquía de memoria

- De más rápida a más lenta: Registros → Caché (L1, L2, L3) → RAM → SSD/HDD → Red/Nube.
- Compromiso entre velocidad, capacidad y coste.

6.2 Características técnicas

- Dirección (32 o 64 bits), latencia, ancho de banda, longitud de palabra.

6.3 Tipos de memoria

- **RAM**: volátil, acceso aleatorio. DRAM (principal), SRAM (caché).
- **ROM**: no volátil. PROM, EPROM, EEPROM.
- **Flash**: persistente, usada en SSD y BIOS.

6.4 Tendencias

- Memoria virtual: paginación, segmentación, swapping.
- Nuevas tecnologías: HBM, GDDR6, Intel Optane, memoria unificada en SoCs (Apple M1/M2).

7. Subsistema de Entrada/Salida (E/S)

7.1 Direccionamiento

- Mapa unificado: memoria y E/S comparten espacio.
- Mapa separado: espacio distinto para cada uno.

7.2 Modos de transferencia

- Por programa (polling): CPU consulta el periférico.
- Por interrupciones: periférico avisa a la CPU.
- DMA (Acceso Directo a Memoria): transfiere sin intervención de la CPU.

7.3 Tecnologías actuales

- USB 4, PCIe 5.0, NVMe, Thunderbolt 4.
- Transmisión síncrona y asíncrona.

8. Buses del sistema

8.1 Tipos

- **Bus de datos:** transporta los datos.
- **Bus de direcciones:** localiza posiciones en memoria.
- **Bus de control:** envía señales de sincronización.

8.2 Clasificación

- Internos: dentro del procesador.
- Externos: entre CPU, memoria y periféricos.

8.3 Temporización

- **Síncrona:** usa reloj común.
- **Asíncrona:** sin reloj compartido.
- Ejemplos modernos: PCIe, USB 4, NVMe.

9. Ciclo de instrucción y ejecución

9.1 Formato de instrucción

- Instrucción = opcode + operandos.
- Puede ser de formato fijo (RISC) o variable (CISC).

9.2 Fases del ciclo

1. Fetch (captura).
2. Decode (decodificación).
3. Execute (ejecución).
4. Memory Access (acceso a memoria).
5. Write Back (escritura del resultado).

9.3 Técnicas de optimización

- **Pipeline:** ejecución en paralelo de fases del ciclo.
- **Superescalaridad:** múltiples instrucciones por ciclo.
- **Ejecución fuera de orden (OoOE).**
- **Predicción de saltos.**
- **Multicore y paralelismo.**
- **Hyper-Threading (SMT).**
- **Procesamiento en GPU.**

10. Futuro de la arquitectura computacional

10.1 Computación cuántica

- Usa qubits: representan simultáneamente 0 y 1.
- Aplicaciones en problemas complejos (factorización, simulaciones).
- Ejemplos: Google Sycamore, IBM Quantum.

10.2 Arquitecturas neuromórficas

- Imitan el cerebro humano (neuronas artificiales).
- Bajísimo consumo, ideales para IA adaptable.
- Ejemplo: Intel Loihi, IBM TrueNorth.

10.3 Chips especializados para IA

- **TPUs (Google):** para redes neuronales y tensores.
- **NPU (móviles):** reconocimiento facial, lenguaje natural.

10.4 Sistemas heterogéneos

- Combinan CPU + GPU + aceleradores (FPGAs, TPUs).
- Alta eficiencia en tareas mixtas.
- Usados en supercomputación, videojuegos, vehículos autónomos y centros de datos.

11. Conclusión

La arquitectura de un ordenador determina su funcionamiento interno, eficiencia y capacidad de adaptación. De las arquitecturas secuenciales clásicas se ha evolucionado hacia modelos paralelos, heterogéneos y especializados, con una mirada hacia el futuro: computación cuántica, inteligencia artificial y nuevos modelos bioinspirados

Actividad: “Diseña tu propio procesador: comprendiendo la arquitectura de un ordenador”

Idea de la actividad:

El alumnado trabajará en grupos para **diseñar un esquema funcional simplificado de un ordenador digital** basado en los modelos arquitectónicos vistos (Von Neumann, Harvard, MIMD, etc.). Cada grupo recibirá un conjunto de requisitos (tipo de arquitectura, número de núcleos, jerarquía de memoria, sistema de E/S, tipo de buses, etc.) y deberá elaborar:

- Un **diagrama funcional** de la arquitectura propuesta (a mano o en herramientas como Lucidchart o Draw.io).
- Una **descripción escrita** del rol de cada componente (CPU, memoria, E/S, buses, etc.).
- Una **justificación técnica** de por qué han escogido esa arquitectura para el escenario propuesto (ej. sistema empotrado, servidor, consola de videojuegos, etc.).

Finalmente, expondrán sus diseños al resto de la clase, explicando cómo se comunican los elementos y qué ventajas ofrece su modelo. La actividad permite aplicar conceptos teóricos de arquitectura desde un enfoque práctico y razonado, desarrollando también competencias de comunicación, síntesis y trabajo en equipo.

Tema 3: Componentes, estructura y funcionamiento de la Unidad Central de Proceso (CPU)

1. Introducción

La Unidad Central de Proceso (CPU) es el componente esencial de un ordenador, encargado de ejecutar instrucciones, realizar operaciones aritmético-lógicas y coordinar todos los procesos del sistema. Su evolución ha sido clave para el desarrollo de la informática moderna.

- **Evolución histórica:**
 - **Mononúcleo:** CPUs clásicas como Intel 8086 o Pentium.
 - **Multinúcleo:** mejora de rendimiento con procesadores como Core 2 Duo o AMD Ryzen.
 - **Arquitecturas híbridas:** combinación de núcleos de alto rendimiento y eficiencia (Intel Alder Lake, ARM big.LITTLE).
- **Tendencias actuales:**
 - Integración de **instrucciones para IA** (Intel DL Boost, Apple Neural Engine).
 - **Arquitecturas heterogéneas:** CPU + GPU integradas en chips como Apple M1/M2.
 - Búsqueda de **eficiencia energética sin pérdida de rendimiento**, fundamental en móviles, portátiles y servidores.

2. Estructura interna de la CPU

2.1 Unidad Aritmético-Lógica (ALU)

- Realiza operaciones matemáticas y lógicas.
- Usa registros internos: acumulador, operandos, flags.
- Incluye unidades especializadas:
 - **FPU (Floating Point Unit):** operaciones en coma flotante.
 - **SIMD / AVX / SSE:** procesamiento vectorial paralelo (clave en gráficos, IA y multimedia).
 - *Ejemplo:* AVX-512 (Intel), NEON (ARM).

2.2 Unidad de Control (UC)

- Gestiona la ejecución de instrucciones: decodifica y emite señales de control.
- Tipos de implementación:
 - **Cableada:** más rápida, menos flexible.
 - **Microprogramada:** más adaptable y actualizable.
- Técnicas modernas:
 - **Pipelining:** solapa la ejecución de instrucciones.
 - **Ejecución especulativa y paralelismo a nivel de instrucción (ILP).**
 - **Predicción de saltos:** cada vez más apoyada en IA para anticipar bifurcaciones en el flujo de ejecución.

2.3 Memoria interna de la CPU

2.3.1 Registros

- Memoria ultrarrápida dentro del procesador.
- Tipos:
 - **Generales:** almacenamiento temporal de datos.
 - **Especiales:** PC (program counter), IR (registro de instrucción), FLAGS (estado), MAR/MDR (dirección/datos de memoria).

2.3.2 Memoria caché

- Reduce la latencia al acceder a datos sin ir a la RAM.
- **Niveles:**
 - **L1:** más rápida, pequeña y específica por núcleo.
 - **L2:** intermedia, compartida por algunos núcleos.
 - **L3:** común a toda la CPU, mayor capacidad.
- Técnicas avanzadas:
 - **Prefetching:** anticipación de datos.
 - **Coherencia de caché:** evita conflictos entre núcleos.
 - *Tendencia:* cachés adaptativas (ej. Intel Adaptive Boost).

2.3.3 Memoria RAM

- Área de trabajo de la CPU.
- Tipos actuales:
 - DDR5 (ordenadores de alto rendimiento).
 - LPDDR5X (dispositivos móviles de bajo consumo).

2.4 Buses internos

- **Bus de datos:** transporta información.
- **Bus de direcciones:** ubica la memoria.
- **Bus de control:** coordina la comunicación interna.
- Evolución:
 - De **FSB (Front-Side Bus)** a tecnologías como **QPI (Intel)**, **Infinity Fabric (AMD)**, **NVLink (NVIDIA)**.
 - *Tendencia:* conexiones internas ultrarrápidas con GPU/memoria (ej. Apple Unified Memory, AMD 3D V-Cache).

3. Funcionamiento de la CPU

3.1 Conjunto de instrucciones

- **CISC (Complex Instruction Set Computing):**
 - Instrucciones complejas.
 - Arquitecturas: x86, ARMv8-A.
- **RISC (Reduced Instruction Set Computing):**
 - Instrucciones simples, más eficientes.
 - Arquitecturas: ARM, RISC-V, Apple Silicon.
- **Extensiones modernas:**
 - **AVX-512:** optimización en IA, multimedia, ciencia de datos.
 - **Intel VT-x / AMD-V:** soporte nativo para virtualización.

Tendencia destacada: adopción creciente de **RISC-V**, una arquitectura abierta y modular.

3.2 Ciclo de instrucción

Fases fundamentales:

1. **Fetch:** búsqueda de la instrucción en memoria.
2. **Decode:** decodificación y preparación.
3. **Execute:** ejecución mediante la ALU o FPU.
4. **Memory Access:** acceso a memoria (si es necesario).
5. **Write-back:** escritura del resultado.

Optimizaciones actuales:

- **Ejecución fuera de orden (OoO).**
- **Predicción de saltos.**
- **Hyper-Threading (Intel) / Simultaneous Multithreading (AMD).**
- *Novedad:* procesamiento de IA en la propia CPU (Intel AMX, Apple Neural Engine).

Conclusión

La CPU ha evolucionado de un diseño monolítico a estructuras complejas y especializadas que integran múltiples núcleos, instrucciones vectoriales, inteligencia artificial y memoria interna avanzada. La tendencia actual se orienta hacia la integración, eficiencia energética y rendimiento en paralelo, lo que redefine la forma en que se diseñan y optimizan los sistemas computacionales en todos los ámbitos, desde dispositivos móviles hasta servidores de alto rendimiento.

Actividad: “Radiografía de una CPU: descubre su estructura y funcionamiento”

Idea de la actividad:

El alumnado realizará una investigación guiada y un modelo explicativo interactivo sobre los componentes internos de una CPU y su funcionamiento. Organizados en grupos, cada equipo se centrará en una parte clave de la CPU (ALU, unidad de control, caché, registros, buses, etc.) y elaborará una presentación visual (física o digital) que explique su función, interacción con otros elementos y relevancia en el ciclo de instrucción.

Además, cada grupo representará de forma práctica una **simulación simplificada del ciclo de instrucción**, asignando roles a los distintos elementos de la CPU para visualizar cómo se procesa una instrucción desde que se busca en memoria hasta que se ejecuta y se guarda el resultado. Esta dinámica servirá para consolidar los conceptos abstractos a través de la experiencia directa y el trabajo colaborativo.

Tema 4: Memoria Interna: Tipos, Direccionamiento, Características y Funciones

1. Introducción

La memoria es un componente esencial en la arquitectura de un ordenador, ya que almacena datos e instrucciones de forma temporal o permanente. Su velocidad, capacidad y organización influyen directamente en el rendimiento global del sistema. Un acceso lento a la memoria puede convertirse en un cuello de botella crítico para la CPU.

2. Conceptos fundamentales de memoria

2.1 Elementos clave

- **Soporte físico:**
 - Silicio: RAM, Flash.
 - Magnético: HDD.
 - Óptico: CD/DVD.
- **Modo de acceso:**
 - Aleatorio (RAM, SSD),
 - Secuencial (cintas),
 - Asociativo (caché).
- **Volatilidad:**
 - Volátil (pierde contenido al apagarse): RAM.
 - No volátil: Flash, HDD, ROM.

2.2 Direccionamiento

- **Direccionamiento bidimensional (2D):** un solo decodificador, usado en memorias pequeñas.
- **Direccionamiento tridimensional (3D):** múltiples decodificadores; se usa en memorias de gran capacidad y alto rendimiento.

2.3 Características clave

- **Velocidad:** medida por latencia y ancho de banda.
- **Unidad de transferencia:** palabra, bloque, línea de caché.
- **Modos de direccionamiento lógico:** directo, indirecto, paginado, segmentado.

3. Tipos de memoria

3.1 Memorias volátiles (almacenamiento temporal)

- **SRAM:** rápida, costosa, sin necesidad de refresco. Se usa en caché.
- **DRAM:** necesita refresco, más lenta pero más densa.
- **SDRAM y DDR (DDR1–DDR5):** sincronizadas con el bus de memoria, más rápidas y eficientes.
- **GDDR5/GDDR6X:** alta capacidad de ancho de banda, usadas en GPUs.

- **HBM (High Bandwidth Memory):** apilamiento vertical para alto rendimiento en IA y servidores.

3.2 Memorias no volátiles (almacenamiento permanente)

- **ROM:** solo lectura, usada para firmware.
- **Flash (NAND/NOR):** base de SSD y dispositivos portátiles.
- **NVRAM:** combina persistencia con velocidad tipo RAM.

4. Jerarquía de memorias y funciones

Organización por velocidad, capacidad y coste:

1. **Registros:** en la CPU, acceso inmediato.
2. **Memoria caché (L1, L2, L3):** reduce latencia de acceso a RAM.
3. **Memoria principal (RAM):** almacena datos activos en ejecución.
4. **Almacenamiento secundario (SSD, HDD):** datos persistentes y programas.
5. **Almacenamiento terciario/red/nube:** respaldo y acceso remoto.

5. Memoria principal y conexión con la CPU

5.1 Estructura física

- **SRAM:** celda con biestables, rápida y costosa.
- **DRAM:** celda con condensador, requiere refresco periódico.
- **ROM:** direccionamiento fijo, contenido no modificable (o solo por reprogramación específica).

5.2 Acceso a memoria

- **Buses involucrados:**
 - Bus de direcciones
 - Bus de datos
 - Bus de control
- **Modos de acceso:**
 - Lectura / Modificación / Escritura
 - Paginación, acceso por columna, y técnicas de acceso rápido.
- **Refresco de DRAM:** distribuido o en ráfagas.

6. Técnicas de mejora del rendimiento

6.1 Memoria caché

- **Tipos:**
 - L1 (más rápida, por núcleo),
 - L2 (más capacidad, por núcleo o compartida),
 - L3 (compartida por todos los núcleos).
- **Mapeos:**
 - Directo, totalmente asociativo, por conjuntos (conjunto asociativo).

- **Políticas de reemplazo:** LRU (menos usado recientemente), FIFO, aleatorio.

6.2 Memoria virtual

- Permite simular más memoria principal utilizando almacenamiento secundario.
- **Traducción de direcciones:**
 - Unidad MMU convierte direcciones virtuales en físicas.
- **Técnicas:**
 - **Paginación:** La memoria se divide en bloques fijos llamados páginas (para procesos) y marcos (en la RAM), lo que evita la fragmentación externa aunque no respeta la estructura lógica del programa; por ejemplo, un proceso de 12 KB se divide en 3 páginas de 4 KB asignadas a marcos libres.
 - **Segmentación:** La memoria se organiza en bloques lógicos como código, datos o pila, lo que permite mantener la estructura del programa aunque puede generar fragmentación externa; por ejemplo, un programa con segmentos de 10 KB, 8 KB y 4 KB se almacena en zonas distintas.
 - **Segmentación paginada:** Cada segmento lógico del programa se subdivide en páginas, combinando el respeto por la estructura del programa con una gestión eficiente del espacio, como cuando el segmento de datos se divide en páginas que se asignan dinámicamente a marcos de memoria.
- **TLB (Translation Lookaside Buffer):** memoria caché para acelerar la traducción de direcciones virtuales.

7. Tecnologías modernas de memoria

7.1 Memoria persistente (PMEM)

- Ejemplo: **Intel Optane**.
- Funciona como RAM, pero conserva los datos tras apagado.

7.2 Memoria apilada 3D (3D XPoint)

- Mayor densidad y baja latencia.
- Mejora el acceso aleatorio respecto a NAND convencional.

7.3 Memoria computacional (PIM – Processing In Memory)

- Procesamiento se realiza dentro del propio chip de memoria.
- Reduce el movimiento de datos → mejora rendimiento y eficiencia.
- Aplicaciones: IA, HPC (computación de alto rendimiento).

Actividad: “Exploradores de la memoria: construyendo la jerarquía desde dentro”

Idea de la actividad:

El alumnado trabajará en equipos para **recrear de forma práctica y visual la jerarquía de memoria de un ordenador**, explicando el papel, características y funcionamiento de cada tipo de memoria (registros, caché, RAM, almacenamiento, etc.). Cada grupo representará un nivel concreto de la jerarquía, diseñará un panel informativo con ejemplos reales (DDR5, SSD, Optane...), incluirá simulaciones o casos de uso, y mostrará cómo se comunica con el resto del sistema.

Además, mediante una dinámica de simulación tipo “cadena de procesamiento”, los grupos representarán el recorrido de una instrucción desde que es buscada en la memoria hasta que se ejecuta, simulando accesos, latencias, sustituciones de caché, y pasos por memoria virtual.

Esta actividad refuerza los conceptos teóricos mediante la visualización, el trabajo colaborativo y la conexión entre arquitectura, rendimiento y tecnología actual.

Tema 10: Representación Interna de los Datos

1. Introducción

Los ordenadores trabajan internamente en **sistema binario (base 2)**. Sin embargo, para distintas necesidades de procesamiento, lectura o comunicación, se utilizan otras bases:

- **Binario (base 2)**: nivel electrónico, representación física.
- **Octal (base 8)**: representación compacta en sistemas antiguos.
- **Decimal (base 10)**: interfaz humana.
- **Hexadecimal (base 16)**: lectura compacta de direcciones, colores, instrucciones.

El conocimiento de estas bases y sus conversiones es esencial en programación, redes y sistemas digitales.

2. Representación de caracteres alfanuméricos

Los caracteres (letras, símbolos, dígitos) se representan mediante códigos binarios estandarizados.

- **ASCII (7 u 8 bits)**: estándar clásico, limitado al inglés.
- **EBCDIC**: desarrollado por IBM, en desuso.
- **UNICODE**:
 - Codificaciones: UTF-8, UTF-16, UTF-32.
 - Soporta todos los idiomas, símbolos científicos y emojis.
 - *UTF-8 es el más utilizado en la web por su eficiencia y compatibilidad*

3. Representación de datos booleanos

- Se representan mediante un solo bit:
 - 0 = falso
 - 1 = verdadero
- Usados en **álgebra de Boole**, lógica digital, condiciones de programación y puertas lógicas.
- **Mapas de Karnaugh**: técnica de simplificación lógica usada en diseño de circuitos y microcontroladores.

4. Representación de números enteros

4.1 Métodos de codificación

- **Signo y magnitud**: bit más significativo representa el signo. Inconveniente: doble representación del 0.
- **Complemento a 1 (CA1)**: mejora anterior, pero mantiene doble 0.
- **Complemento a 2 (CA2)**:
 - Estándar en sistemas actuales.
 - Simplifica la resta y el tratamiento de negativos.
 - Ejemplo en 8 bits: $-1 = 11111111$, $1 = 00000001$.

4.2 Representación en exceso-Z

- Se utiliza en exponentes de coma flotante (IEEE 754).
- Permite trabajar con exponentes negativos mediante desplazamiento.

5. Representación de números reales

5.1 Formatos

- **Coma fija:** poco usado hoy por su precisión limitada.
- **Coma flotante (IEEE 754):** estándar internacional.
 - **32 bits (simple precisión):** uso general.
 - **64 bits (doble precisión):** cálculo científico.
 - **128 bits (cuádruple precisión):** supercomputación.

5.2 Conceptos clave

- **Normalización:** formato estándar que maximiza precisión.
- **Exponente y mantisa:** permiten representar números muy grandes o muy pequeños.
- **Desbordamiento / subdesbordamiento:** errores por superar o no alcanzar los límites de representación.

6. Representación de números complejos

- Se representan mediante dos componentes en coma flotante:
 - Parte real + parte imaginaria.
- Aplicaciones:
 - **Procesamiento de señales:** audio, telecomunicaciones.
 - **Computación cuántica:** amplitudes de probabilidad.
 - **Gráficos 3D y simulaciones físicas.**

7. Representación interna de estructuras de datos

7.1 Estructuras lineales

- **Vectores y matrices:** acceso indexado por posición.
- **Listas enlazadas:** nodos conectados dinámicamente; eficientes en inserciones/borrados.

7.2 Estructuras jerárquicas

- **Árboles:**
 - **BST:** árbol binario de búsqueda.
 - **AVL, B+, B:*** árboles balanceados, usados en bases de datos y sistemas de archivos.

7.3 Grafos

- Representación:
 - **Listas de adyacencia:** más eficiente en espacio.

- **Matrices de adyacencia:** acceso rápido.
- Aplicaciones: redes, mapas GPS, algoritmos de inteligencia artificial.

7.4 Tablas hash

- Permiten búsqueda casi constante: $O(1)$.
- Uso: bases de datos, sistemas de caché, compiladores.

7.5 Punteros y estructuras dinámicas

- Uso esencial en C/C++.
- Permiten manipular directamente la memoria, crear estructuras enlazadas y gestionar almacenamiento dinámico.

8. Representación de elementos multimedia

8.1 Imagen

- **Gráficos vectoriales:** SVG, PDF. Escalables sin pérdida de calidad.
- **Mapas de bits (raster):** BMP, PNG, JPEG, WebP.
- **Compresión:**
 - *Con pérdida:* JPEG, HEIC → menor tamaño.
 - *Sin pérdida:* PNG → mantiene calidad.

8.2 Sonido

- Representación digital por muestreo.
- Formatos: WAV (sin compresión), MP3, OGG, FLAC.
- Parámetros: frecuencia de muestreo, resolución (bits).

8.3 Video

- Codificación por frames e interpolación.
- Formatos: MPEG, MP4, H.265.
- Códecs actuales optimizan calidad y compresión para streaming.

8.4 Gráficos 3D

- Modelado de objetos con vértices, texturas y materiales.
- Formatos: OBJ, FBX, GLTF.
- Aplicaciones: videojuegos, realidad virtual, CAD.

9. Cifrado y compresión

9.1 Cifrado

- **AES:** cifrado simétrico moderno y rápido.
- **RSA:** cifrado asimétrico basado en números primos grandes.
- **ECC (criptografía de curvas elípticas):** menor tamaño de clave, mismo nivel de seguridad.

- **Criptografía post-cuántica:** en desarrollo para resistir ataques de ordenadores cuánticos.

9.2 Compresión

- **Sin pérdida:** ZIP, PNG, FLAC. Recuperación exacta.
- **Con pérdida:** MP3, JPEG, H.265. Elimina información no esencial para reducir tamaño.

9.3 Funciones hash

- Códigos únicos generados a partir de datos.
- Aplicaciones:
 - Integridad de datos.
 - Autenticación.
 - Indexación rápida.

Actividad: “Del bit al mundo: cómo representa el ordenador todo lo que ves”

Idea de la actividad:

El alumnado realizará un recorrido práctico por los distintos tipos de datos que maneja un ordenador, representando cada uno internamente en binario. Divididos por equipos, cada grupo se encargará de **representar un tipo de dato** (caracteres, enteros, reales, estructuras, imágenes, sonido, etc.) con ejemplos concretos que luego explicarán al resto de la clase.

Cada grupo deberá:

- Investigar cómo se codifica su tipo de dato (formato, bits, estándares).
- Desarrollar un **ejemplo práctico** (por ejemplo, codificar una palabra en ASCII/UTF-8, una imagen en mapa de bits simplificado, una canción con parámetros de sonido, un número negativo en complemento a 2, etc.).
- Representar visualmente esa codificación (tablas, gráficos, bloques binarios).
- Preparar una **breve exposición** explicando la conversión del dato desde lo humano a lo binario y viceversa.

Opcionalmente, los grupos pueden simular el **almacenamiento, compresión o cifrado** de su dato con herramientas digitales simples (calculadoras binarias, editores hexadecimales, convertidores de codificación).

Esta actividad permite **integrar teoría con práctica**, trabajar la lógica binaria, reconocer la importancia del formato en programación y visualizar cómo un sistema digital representa cualquier tipo de información.

Tema 11: Organización Lógica de los Datos. Estructuras Estáticas

1. Introducción

La **organización lógica de los datos** es la base sobre la que se construyen algoritmos y estructuras en programación. Consiste en definir cómo se agrupan, relacionan y manipulan los datos desde una perspectiva abstracta, sin depender de su implementación física en memoria.

- **Abstracción de datos:** separación entre la representación lógica (qué se hace) y la física (cómo se implementa).
- **Tipos de datos:** conjunto de valores posibles junto con operaciones definidas sobre ellos.
- **Importancia:** permite diseñar algoritmos correctos, eficientes y reutilizables.

2. Tipos Abstractos de Datos (TAD)

2.1 Definición y componentes

Un **TAD** es un modelo lógico que describe un conjunto de datos y sus operaciones, **independiente de la implementación**.

- **Componentes de un TAD:**
 - Conjunto de datos.
 - Operaciones posibles.
 - Propiedades semánticas (reglas que deben cumplir las operaciones).

2.2 Ejemplos comunes de TAD

- **Pila (Stack):** LIFO (Last In, First Out).
- **Cola (Queue):** FIFO (First In, First Out).
- **Lista:** secuencia ordenada, permite inserción/borrado.
- **Árbol:** estructura jerárquica (padre-hijo).
- **Grafo:** modela relaciones complejas, como redes.
- **Tabla hash:** permite acceso rápido mediante clave.

3. Tipos de datos escalares

3.1 Tipos normalizados

- **Entero:** representado en complemento a 2.
- **Real:** estándar IEEE 754 (simple, doble precisión).
- **Carácter:** codificado en Unicode (UTF-8, UTF-16).
- **Booleano:** representa verdadero o falso (0 / 1).

3.2 Tipos definidos por el usuario

- **Enumeración:** conjunto cerrado de valores (ej. {Rojo, Verde, Azul}).

- **Rango:** subconjunto continuo de un tipo base (ej. `1..100`).

4. Tipos de datos estructurados

4.1 Vectores (Arrays)

- **Unidimensionales:** útiles para cadenas, listas simples.
- **Multidimensionales:** representación de matrices, imágenes, juegos de datos complejos.

4.2 Conjuntos

- Operaciones básicas: unión, intersección, diferencia.
- Eficientes en programación lógica y matemática.

4.3 Registros y tuplas

- **Registros (struct):** agrupan varios tipos de datos con nombre.
- **Variantes:** permiten estructuras flexibles (como `union` en C).
 - Parte fija + parte variable según un selector.

5. Implementación estática de estructuras de datos

Uso de arrays como base de almacenamiento; se conoce el tamaño de antemano.

5.1 Pilas (Stacks)

- Implementación: array + puntero al tope.
- Operaciones: `push()`, `pop()`, `top()`.
- Usos: llamadas a funciones, expresiones, backtracking.

5.2 Colas (Queues y Deques)

- **Colas simples:** FIFO, array circular.
- **Deques:** permiten inserciones y eliminaciones por ambos extremos.
- Usos: planificación de procesos, estructuras reactivas.

5.3 Listas

- Comparativa:
 - **Listas enlazadas:** dinámicas, flexibles.
 - **Arrays:** rápidos en acceso indexado.
- Inserciones/borrados costosos en arrays estáticos.

5.4 Árboles

BST (Árbol Binario de Búsqueda)

- Nodo: máx. 2 hijos
- Izquierda < nodo < derecha

- Búsqueda eficiente ($O(\log n)$), pero puede desbalancearse

Balanceados (AVL, B+)

- Mantienen equilibrio automáticamente
- Operaciones siempre $O(\log n)$
- B+: usado en bases de datos

Heap (Montículo)

- Árbol binario completo
- Max-heap: padres \geq hijos / Min-heap: padres \leq hijos
- Implementado en arrays ($i \rightarrow 2i+1, 2i+2$)
- Usado en colas de prioridad y heapsort

Usos

- Bases de datos, compiladores, sistemas jerárquicos

5.5 Grafos

- Representación:
 - **Matriz de adyacencia:** más memoria, acceso inmediato.
 - **Lista de adyacencia:** menos memoria en grafos dispersos.
- Aplicaciones: redes, mapas, algoritmos como Dijkstra o A*.

5.6 Tablas hash

- Implementación: array indexado por función hash.
- Gestión de colisiones: encadenamiento o direccionamiento abierto.
- Usos: bases de datos, autenticación, almacenamiento en caché.

6. Aplicación práctica: Concursos y entrenamiento algorítmico

6.1 Por qué es útil programar con estructuras estáticas

- Permiten modelar y resolver problemas reales de forma eficiente.
- Su limitación de crecimiento en muchos contextos es una virtud y no un problema para dotar de estabilidad al sistema.
- Fomentan el pensamiento lógico y la abstracción.
- Su dominio es clave en entrevistas técnicas y competiciones.

6.2 Competiciones relevantes

- **Olimpiada Informática Española (OIE):**
 - Nivel preuniversitario.
 - Fases: regional, nacional e internacional (IOI).
 - Enfoque: algoritmos y estructuras eficientes.
- **ProgramaMe:**
 - Concurso nacional para FP.
 - Modalidad por equipos, pruebas de eficiencia y estructuras.

6.3 Recursos para el entrenamiento

- Plataformas online:
 - **Codeforces, LeetCode, AtCoder, HackerRank.**
- Mejora la agilidad mental, el manejo de estructuras y la optimización del código.

Actividad: “Diseña y defiende tu estructura”

Idea de la actividad:

El alumnado, organizado por grupos, investigará y seleccionará una **estructura de datos estática** (como pila, cola, vector, árbol, grafo o tabla hash) para **modelar un problema cotidiano o técnico** que pueda resolverse con dicha estructura. Cada grupo deberá:

1. **Justificar la elección** de la estructura según sus propiedades lógicas (orden, acceso, inserción, búsqueda, etc.).
2. **Diseñar un modelo visual** (diagrama o simulación simple) que explique su funcionamiento.
3. **Definir las operaciones básicas** de la estructura con pseudocódigo o esquemas paso a paso.
4. **Relacionarla con un caso real o aplicado**, como puede ser el historial de un navegador (pila), la gestión de procesos en una impresora (cola), o la planificación de rutas (grafos).
5. **Presentar su trabajo oralmente** defendiendo por qué su estructura es la más adecuada para el problema asignado.

La actividad busca reforzar la comprensión de los TADs y estructuras estáticas desde la lógica y el razonamiento algorítmico, promoviendo el trabajo colaborativo, el diseño orientado a problemas y la exposición técnica.

Tema 12: Organización Lógica de los Datos – Estructuras Dinámicas

1. Introducción

Las **estructuras dinámicas** permiten gestionar datos sin conocer de antemano su tamaño, adaptándose al crecimiento o reducción del contenido durante la ejecución de un programa.

- A diferencia de las estructuras estáticas, **no tienen tamaño fijo**.
- Se apoyan en **punteros o referencias** para enlazar nodos en memoria.
- Facilitan la **inserción, eliminación y reorganización eficiente** de elementos.

2. Fundamentos de las estructuras dinámicas

2.1 Características clave

- Gestión de memoria en tiempo de ejecución.
- Flexibilidad estructural.
- Uso intensivo de punteros o referencias.

2.2 Ventajas

- Eficiencia en operaciones frecuentes de inserción y borrado.
- Utilidad en contextos donde los datos cambian de tamaño con frecuencia.

2.3 Inconvenientes

- Mayor complejidad de implementación.
- Coste adicional en gestión de memoria y recorrido.
- Acceso más lento que en arrays.

3. Listas dinámicas

3.1 Listas enlazadas simples

- Cada nodo contiene:
 - Dato.
 - Puntero al siguiente nodo.
- Operaciones: insertar, eliminar, recorrer, buscar.
- Usos: almacenamiento flexible, estructuras auxiliares.

3.2 Listas doblemente enlazadas

- Cada nodo tiene puntero al **siguiente** y al **anterior**.
- Permiten navegación en ambos sentidos.
- Útiles para implementaciones de deque o editores de texto.

3.3 Listas circulares

- El último nodo apunta al primero.
- Evitan referencias nulas, útiles en aplicaciones cíclicas como planificadores.

4. Pilas y colas dinámicas

4.1 Pilas

- Implementadas con listas enlazadas.
- Modelo LIFO (último en entrar, primero en salir).
- Operaciones: `push()`, `pop()`, `top()`.

4.2 Colas

- Modelo FIFO (primero en entrar, primero en salir).
- Operaciones: `enqueue()`, `dequeue()`, `front()`.

4.3 Colas dobles (deques)

- Inserción/eliminación por ambos extremos.
- Versátiles para algoritmos de recorrido y planificación.

5. Árboles dinámicos

5.1 Árbol binario

- Cada nodo tiene como máximo dos hijos: izquierdo y derecho.
- Operaciones: inserción, recorrido (inorden [izq – nodo – der (ordenado)], preorden [nodo – izq – der (estructura)], postorden[izq – der – nodo (eliminación)]), búsqueda.

5.2 Árbol binario de búsqueda (BST)

- Ordenado: nodo izquierdo < nodo < nodo derecho.
- Permite búsqueda eficiente si está equilibrado.

5.3 Árboles balanceados

- **AVL**: se mantiene balanceado tras cada inserción/eliminación.
- **Red-Black**: garantiza balanceo con menor coste computacional.
- Mejoran el rendimiento en inserciones y búsquedas.

5.4 Árboles n-arios y generalizados

- Permiten más de dos hijos por nodo.
- Usos: árboles de expresión, jerarquías organizativas, XML, DOM.

5.5 Heap (Montículo)

Árbol binario completo con orden específico:

- **Max-heap**: padres \geq hijos

- **Min-heap:** padres \leq hijos
Implementado en arrays ($i \rightarrow 2i+1, 2i+2$)
Usos: colas de prioridad, heapsort, planificación de procesos

6. Grafos dinámicos

6.1 Representación mediante listas de adyacencia

- Cada nodo tiene lista con sus conexiones.
- Más eficiente en espacio para grafos dispersos.

6.2 Nodos enlazados

- Cada vértice enlaza a sus aristas.
- Pueden representar grafos dirigidos o no dirigidos, con pesos o sin ellos.

6.3 Aplicaciones

- Redes de comunicación, redes sociales, rutas GPS, IA.

7. Tablas hash con listas de colisiones

- Resolución de colisiones mediante **encadenamiento**.
- Cada posición del array apunta a una **lista enlazada** de entradas con la misma clave hash.
- Mejora eficiencia en inserciones múltiples.
- Usos: diccionarios, cachés, bases de datos.

8. Gestión dinámica de memoria

8.1 Asignación y liberación

- En C/C++: `malloc`, `free`, `new`, `delete`.
- En Java, Python: manejo automático con recolector de basura.

8.2 Problemas comunes

- **Pérdida de memoria (memory leaks):** olvidarse de liberar memoria.
- **Doble liberación:** intentar liberar la misma zona más de una vez.
- **Fragmentación:** uso ineficiente del espacio de memoria.

9. Aplicaciones y contexto de uso

- **Sistemas operativos:** gestión de procesos, colas de planificación.
- **Compiladores:** árboles de sintaxis abstracta.
- **Editores de texto:** listas enlazadas para líneas o bloques.
- **Juegos y simulaciones:** estructuras de comportamiento dinámico.
- **Bases de datos:** índices dinámicos (árboles B, B+).

10. Conclusión

Las estructuras dinámicas permiten una mayor **adaptabilidad y eficiencia** en programas donde los datos cambian constantemente. Su dominio requiere comprensión de punteros, memoria y algoritmos de recorrido. Son imprescindibles para diseñar software eficiente, seguro y escalable.

Actividad: “Simula una estructura viva: programando con nodos”

Idea de la actividad:

El alumnado desarrollará por equipos una **simulación visual o textual de una estructura de datos dinámica** (como lista enlazada, árbol binario, cola o grafo) que represente un sistema con cambios constantes, como una playlist musical, una cola de impresión, una jerarquía de menús o una red de rutas. El objetivo es que comprendan cómo se comportan los datos **cuando se insertan, eliminan o reorganizan** en memoria mediante enlaces dinámicos.

Tema 13: Ficheros: Tipos, Características, Organizaciones

1. Introducción: El fichero como núcleo del almacenamiento digital

- Un fichero es una unidad lógica de almacenamiento que encapsula información estructurada.
- La gestión eficiente de ficheros es esencial para el rendimiento, la seguridad y la interoperabilidad de los sistemas.
- En entornos actuales, los ficheros no solo contienen datos: **pueden ser modelos, logs, configuraciones, multimedia o flujos de telemetría.**

2. Estructura lógica del fichero

- **Campo:** unidad básica (ej. **nombre**, **precio**).
- **Registro:** conjunto de campos (ej. ficha de producto).
- **Fichero:** colección de registros organizados.

 *Visual:* **nombre**, **precio**, **categoría** → Registro 1: "Ratón", 25, "Electrónica".

3. Tipos de ficheros

a) Por contenido

- **Texto plano** (.txt, .csv): legibles por humanos.
- **Binarios** (.exe, .png): legibles por sistemas.

b) Por función

- **Datos** (log.json, ventas.csv).
- **Código y scripts** (.py, .sh, .jar).
- **Configuración** (.yaml, .ini, .conf).

c) Por formato moderno

- **Documentales:** JSON, BSON (MongoDB).
- **Big Data:** Parquet, Avro, ORC (columnar).
- **Multimedia:** MP4, MP3, WebP, FLAC.

4. Características de los ficheros

- **Nombre y extensión.**
- **Tamaño y peso lógico/físico.**
- **Metadatos:** permisos, timestamps.
- **Ruta y ubicación.**
- **Atributos de sistema:** inmutabilidad, enlaces, cifrado.

5. Organización de ficheros (estructura física)

Organización	Acceso	Rendimiento	Uso típico
Secuencial	Lineal	Bajo	Logs, historiales
Indexada	Parcial aleatorio	Medio	Catálogos, búsqueda parcial
Directa (hash)	Aleatorio directo	Alto	Bases de datos OLTP, caches
Encadenada	Dinámico	Irregular	Sistemas distribuidos, blockchain

6. Sistemas de archivos modernos (clasificados por enfoque)

Seguridad

- **NTFS**: ACL, journaling, cifrado EFS.
- **APFS**: cifrado nativo, snapshots.
- **ZFS**: integridad, replicación, verificación.
- **Btrfs**: snapshots, verificación, compresión.

Rendimiento

- **EXT4**: alta compatibilidad y velocidad.
- **XFS**: I/O intensivo, grandes volúmenes.
- **tmpfs**: ultra rápido, RAM-based.

Distribuidos / Cloud

- **Amazon S3 / Azure Blob**: object storage escalable.
- **HDFS**: clústeres Big Data.
- **CephFS / GlusterFS**: POSIX distribuido, replicado.

Compatibilidad

- **FAT32 / exFAT**: máxima portabilidad entre sistemas.

7. Usos y contextos actuales

- **Sistemas operativos**: configuración, arranque, logs.
- **Bases de datos**: archivos de tabla e índice.
- **Aplicaciones móviles/web**: multimedia, JSON.
- **IoT**: ficheros compactos, binarios de configuración.
- **DevOps y ML**: tracking de modelos, ficheros **.pk1**, **.h5**, **.onnx**.


8. Tendencias tecnológicas

- **Cifrado por defecto y privacidad** (BitLocker, LUKS).
- **Clasificación inteligente de ficheros por IA** (duplicados, clusters).

- **Integración cloud-native:** sync automático, control de versiones.
- **Optimización por uso:** S3 Lifecycle, edge cache, formatos columnar.


9. Cifrado de ficheros y protección

- **Simétrico:** AES-256 en archivos locales o contenedores (VeraCrypt).
- **Asimétrico:** GPG/PGP en correo seguro o firmas.
- **Nivel de sistema:** BitLocker, LUKS, APFS encryption.

 *Ejemplo:* cifrado automático en backups, contratos firmados con clave privada.

10. Control de versiones de ficheros

- **Git:** ideal para texto, config, código.
- **DVC / MLflow:** versionado de datasets y modelos en IA.
- **.gitignore y Git LFS:** gestión de ficheros grandes o binarios.

 *Aplicación práctica:* mantener históricos de scripts, documentación, y configuración de proyectos.

11. Ficheros en la nube: almacenamiento moderno y ubicuo

¿Qué son?

- Ficheros gestionados a través de plataformas remotas accesibles desde internet (SaaS, PaaS).
- Se almacenan en centros de datos distribuidos y son accesibles desde múltiples dispositivos.

Características clave:

- **Accesibilidad ubicua:** desde cualquier dispositivo y lugar.
- **Sincronización automática:** actualizaciones en tiempo real.
- **Versionado:** histórico de cambios, restauración de versiones.
- **Compartición colaborativa:** control granular de acceso, coedición.

Ejemplos de uso:

- **Usuarios personales:** Google Drive, iCloud, Dropbox.
- **Empresas:** OneDrive for Business, Box, Nextcloud.
- **Cloud pública:** Amazon S3, Azure Blob Storage, Google Cloud Storage.

Ventajas:

- Reducción de pérdidas por fallos locales.
- Mejora de la productividad colaborativa.
- Escalabilidad y coste por uso.

Riesgos y contramedidas:

- **Privacidad y dependencia del proveedor** → uso de cifrado, backups locales.
- **Pérdida de control físico** → políticas de seguridad y acceso (Zero Trust, MFA).

12. Conclusión

Los ficheros siguen siendo la **unidad base de persistencia digital**, pero han evolucionado hacia sistemas estructurados, protegidos y colaborativos. Comprender su gestión, organización y contexto es clave para el diseño de sistemas modernos: desde microservicios hasta clústeres distribuidos, desde un servidor local hasta la nube. Su dominio permite crear entornos seguros, rápidos y adaptables, en una era donde el dato es el nuevo activo.

Tema 16: Sistemas Operativos: Gestión de Procesos

1. Introducción

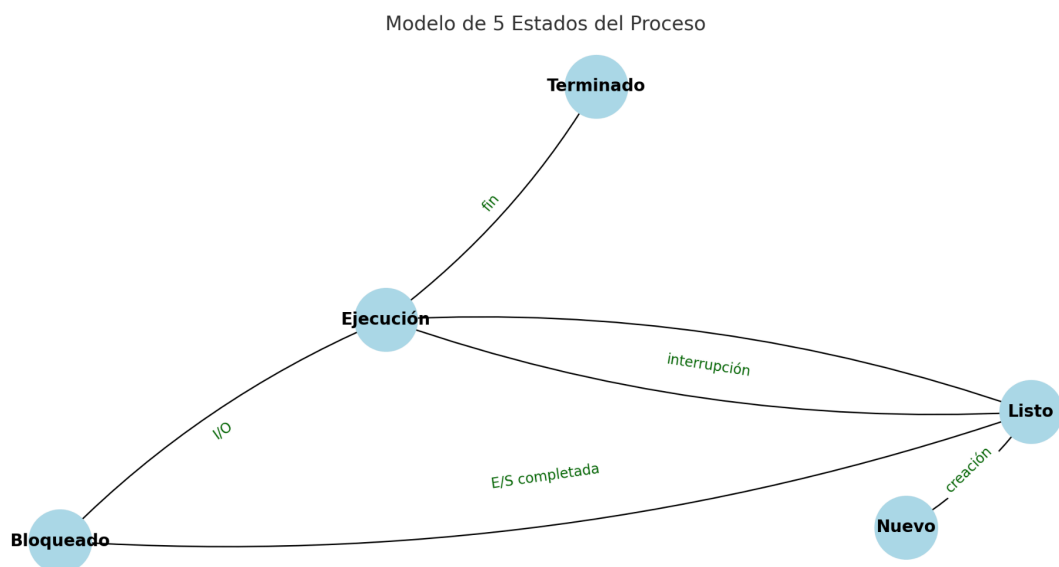
Un **proceso** es un programa en ejecución. No solo ejecuta código, sino que tiene recursos asignados por el sistema operativo: CPU, memoria, archivos, dispositivos.

La **gestión de procesos** permite que múltiples programas convivan sin interferirse: multitarea, eficiencia y aislamiento.

¿Dónde es clave?

- En **servidores cloud**, donde corren decenas de servicios en paralelo.
- En **sistemas de inteligencia artificial**, que requieren entrenamiento y ejecución concurrente, a menudo en GPU.
- En **contenedores y microservicios**, donde cada servicio es un proceso (o grupo) aislado.

2. Concepto y Ciclo de Vida del Proceso (Modelo de 5 Estados Básico)



Un proceso es más que una secuencia de instrucciones: es una entidad dinámica gestionada por el sistema operativo. Se compone de:

- **Código ejecutable** (segmento de texto),
- **Datos** (variables globales, buffers),
- **Pila** (para llamadas a funciones y almacenamiento local),
- **Contexto de ejecución** (registros de CPU, contador de programa).

Modelo clásico de 5 estados:

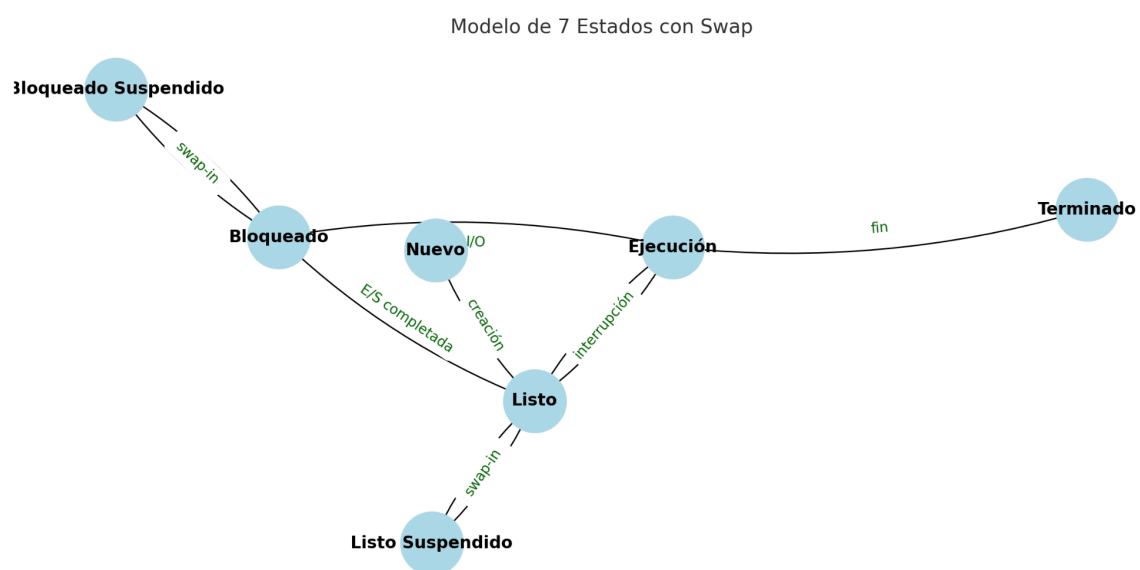
1. **Nuevo** (New): el proceso está siendo creado.

2. **Listo** (*Ready*): ha sido cargado en memoria y espera la CPU.
3. **Ejecución** (*Running*): el proceso está usando la CPU.
4. **Bloqueado** (*Waiting*): espera por E/S u otro recurso.
5. **Terminado** (*Terminated*): ha finalizado su ejecución.

Transiciones destacadas:

- **dispatch**: listo → ejecución.
- **I/O**: ejecución → bloqueado.
- **interrupt**: ejecución → listo.

3. Modelo de 7 Estados con Intercambio de Memoria (Swap)



En sistemas que usan **memoria virtual** o requieren optimización de recursos, se utiliza un modelo más detallado con **7 estados**, que incorpora procesos que están **intercambiados al disco (swap)**.

Estados adicionales:

6. **Listo Suspendido** (*Ready Suspended*):
El proceso estaba en memoria listo para ejecutarse, pero ha sido desplazado a disco para liberar RAM. No puede ejecutarse hasta ser cargado de nuevo.
7. **Bloqueado Suspendido** (*Blocked Suspended*):
Igual que el bloqueado, pero además se ha intercambiado fuera de memoria. Está esperando un recurso y no está en RAM.

Transiciones clave (con swap):

- **Listo → Listo Suspendido**: cuando hay presión de memoria, el sistema intercambia procesos inactivos al disco.

- **Bloqueado** → **Bloqueado Suspendido**: similar, pero el proceso está esperando E/S.
- **Listo Suspendido** → **Listo**: se recupera de disco y vuelve a RAM cuando hay espacio y prioridad.
- **Bloqueado Suspendido** → **Bloqueado**: también vuelve a memoria para reanudar espera activa.
- **Bloqueado** → **Listo**: cuando se completa la E/S.
- **Listo** → **Ejecución**: el planificador le asigna CPU.

Ventajas del modelo extendido:

- Permite **sobrecargar la memoria RAM** sin perder procesos.
- Mejora el **rendimiento global** en entornos con alta concurrencia.
- Es la base de la **memoria virtual moderna**.

Analogía útil para exposición oral:

Piensa en una oficina pequeña (RAM) con muchos empleados (procesos). Cuando está llena, algunos se van al archivo (disco). Siguen empleados, pero no están presentes físicamente. Cuando se les necesita, vuelven al escritorio.

4. Bloque de Control del Proceso (PCB)

El **PCB** es la estructura de datos donde el sistema operativo guarda toda la información de un proceso:

- Identificador (PID).
- Estado.
- Registros del procesador.
- Punteros a memoria y archivos abiertos.
- Estadísticas: tiempo en CPU, uso de E/S.
- Seguridad: UID, GID, privilegios.

En Linux se puede consultar esta información en `/proc/[pid]/`.

5. Hilos (Threads)

Un **hilo** es una subunidad de ejecución dentro de un proceso. Todos los hilos del mismo proceso comparten:

- El código.
- El espacio de direcciones (memoria).
- Los archivos abiertos.

Pero **cada hilo mantiene su propio contexto**:

- Contador de programa.
- Registros.
- Pila de ejecución.

Esto permite que múltiples tareas se ejecuten en paralelo dentro del mismo proceso.

Ventajas de usar hilos


- **Ligereza:** consumen menos recursos que los procesos.
- **Rapidez:** el cambio de contexto entre hilos es más eficiente.
- **Paralelismo:** permiten aprovechar todos los núcleos de CPU.


Son ideales para:

- Servidores web que manejan múltiples conexiones.
- Aplicaciones gráficas y móviles.
- Algoritmos de IA que ejecutan múltiples tareas simultáneamente.

Modelos de hilos

Modelo 1:1 (uno a uno)

 **Qué es:** Cada hilo que crea tu programa se convierte en un hilo que el sistema operativo también ve y administra.


 **Ejemplo real:** pthreads en Linux.

 **Ventajas:**


- Se pueden usar varios núcleos: los hilos pueden ejecutarse verdaderamente en paralelo.
- Ideal para tareas intensivas.

 **Desventaja:**

- Más costoso en recursos: cada hilo ocupa memoria y necesita ser gestionado por el sistema.

 **Analogía:** Cada empleado tiene su propia computadora y el jefe (el SO) tiene que vigilar a todos.

Modelo N:1 (muchos a uno)


 **Qué es:** Muchos hilos en tu programa, pero el sistema operativo solo ve uno. Todo se ejecuta en un único hilo del sistema.

 **Ventajas:**


- Muy eficiente y rápido de crear/cambiar entre hilos.
- Menor sobrecarga para el sistema.

 **Desventaja:**

- **No se aprovechan varios núcleos.** Si un hilo se bloquea (espera algo), todos los demás se detienen también.

 **Analogía:** Todos los empleados comparten **una sola computadora**. Solo uno puede usarla a la vez.

Modelo M:N (muchos a muchos)


 **Qué es:** Combina lo mejor de los dos modelos. M hilos de usuario se reparten entre N hilos del sistema.

Ventajas:

- Buena escalabilidad y eficiencia.
- Posibilidad real de paralelismo con flexibilidad.

Desventaja:

- Más difícil de programar y mantener. Requiere una capa intermedia de gestión.

 **Analogía:** Hay muchos empleados y varias computadoras, pero no todos tienen una fija. Se reparten dinámicamente según disponibilidad.

¿Dónde se usan?

Modelo	Usado en...
1:1	Linux (pthreads), Java Threads
N:1	Lenguajes antiguos, algunos entornos educativos
M:N	Go (goroutines), Erlang, sistemas avanzados como Windows Server antiguos

Peligros de la multihilación

- **Condiciones de carrera:** cuando dos hilos acceden al mismo dato sin control.
- **Bloqueos mutuos:** si dos hilos esperan recursos entre sí.
- **Errores difíciles de depurar:** los fallos concurrentes suelen ser aleatorios.

 **Solución:** usar mecanismos como **mutex**, **semáforos** o **monitores** para sincronizar.

6. Planificación de la CPU

El objetivo es decidir **qué proceso se ejecuta, cuándo y por cuánto tiempo**.

Criterios:

- *Turnaround time*: tiempo total desde inicio a fin.
- *Waiting time*: tiempo total esperando en cola.
- *Response time*: tiempo hasta recibir la primera respuesta.
- *Throughput*: procesos completados por unidad de tiempo.

Algoritmos de planificación:

Algoritmo	Tipo	Características	Uso típico
FCFS	No apropiativo	Orden de llegada	Procesamiento por lotes
SJF / SRTF	No / Sí	Jobs cortos primero	Sistemas batch
Round Robin	Apropiativo	Cuanto fijo, rotación circular	Sistemas interactivos
Prioridades	Mixto	Basado en prioridad	Tiempo real, IA, servidores
Multicolas	Avanzado	Separación por tipo de proceso	Cloud, microservicios

Linux usa el **CFS (Completely Fair Scheduler)**, que emplea un árbol **rojo-negro** (estructura balanceada).

Organiza los procesos listos en función del tiempo de uso virtual, permitiendo seleccionar el siguiente en tiempo **$O(\log n)$** .

El **CFS (Completely Fair Scheduler)** es el planificador por defecto en Linux desde el kernel 2.6.23. Su objetivo es **repartir el tiempo de CPU de forma justa entre todos los procesos**, en vez de usar un sistema de colas tradicional con prioridades fijas o quanta estricta como Round Robin.



¿Usa quantum?

No exactamente.

CFS **no usa un quantum fijo** como Round Robin. En lugar de eso:

- Estima cuánto tiempo debería ejecutarse cada proceso según su prioridad y cuánto ha usado ya la CPU.

- Elige siempre el proceso que **menos tiempo ha usado** en relación a lo que le "toca".
- Así logra una especie de "justicia proporcional": todos los procesos obtienen una fracción **equilibrada** de CPU.

📌 Pero... sí puede **limitar** el tiempo máximo de ejecución de un proceso antes de devolver la CPU, para evitar que monopolice recursos. Esto actúa como un *quantum dinámico*, adaptable.

🌳 ¿Cómo lo implementa?

Usa una **estructura de datos muy eficiente**: un **árbol rojo-negro**.

¿Qué es un árbol rojo-negro?

- Es un **árbol binario de búsqueda auto-balanceado**.
- Permite operaciones en **tiempo logarítmico** $O(\log n)$:
 - Insertar un proceso.
 - Eliminarlo cuando termina.
 - Encontrar el siguiente proceso a ejecutar.

⚙️ ¿Qué guarda el árbol?

Cada nodo del árbol representa un **proceso listo para ejecutarse**, y está ordenado por su **tiempo de uso virtual** (**vruntime**):

- Este **vruntime** es una medida ajustada del tiempo real que el proceso ha consumido.
- Los procesos que han usado **menos CPU** aparecen **más a la izquierda**.
- Por tanto, el proceso a ejecutar es siempre el **mínimo** del árbol: el más "justo".



¿Qué lo hace especial?

- Es **completamente equitativo**: ningún proceso se queda sin CPU.
- **Flexible**: adapta el reparto si hay procesos interactivos o en segundo plano.
- **Eficiente**: mantener el árbol balanceado asegura que seleccionar el siguiente proceso sea rápido incluso con miles de procesos.



Ejemplo práctico

Imagina 3 procesos:

- P1 es interactivo (consume poco CPU),
- P2 es un servidor web,
- P3 es un análisis de datos pesado.


CFS observa que P3 consume más CPU, y reduce su **vruntime**, dejando más CPU para P1 y P2. Así:

- P1 responde rápido (latencia baja).
- P3 sigue progresando, pero sin acaparar.

7. Concurrency y sincronización de Procesos

¿Qué es la concurrencia?

La **concurrencia** ocurre cuando múltiples procesos o hilos **ejecutan en paralelo** o **comparten recursos** de forma intercalada. Aunque no todos se ejecuten simultáneamente (como en sistemas mononúcleo), sí pueden avanzar *aparentemente al mismo tiempo*.

 **Problema:** cuando acceden a **recursos compartidos** (como variables, ficheros o buffers), pueden aparecer **condiciones de carrera**, donde el resultado depende del orden de ejecución.

Mecanismos de sincronización

1. Semáforos

Variables especiales que controlan el acceso a recursos.

- *Binarios (mutex)*: acceso exclusivo.
- *De conteo*: controlan múltiples instancias del recurso.
- Operaciones: `wait()` (P) y `signal()` (V). Operaciones con ejecución unitaria.

2. Monitores

Estructuras de más alto nivel que combinan datos y funciones sincronizadas.

- Usan variables de condición internas.
- Gestionan la entrada y salida de procesos automáticamente.

3. Spinlocks

Bloqueos activos: el proceso espera en bucle.

Útiles en sistemas multiprocesador donde el cambio de contexto sería más costoso que el bloqueo.

Problema clásico: los 5 filósofos

Cinco filósofos sentados a una mesa.

Cada uno necesita **dos tenedores** (a su izquierda y derecha) para comer, pero hay solo uno entre cada par.

Todos piensan y quieren comer al mismo tiempo.

El problema:

- Si todos toman primero el tenedor izquierdo, **ninguno podrá tomar el derecho** → interbloqueo.
- Si no se gestiona bien, puede haber **inanición**: uno nunca logra comer.

Soluciones típicas:

- Controlar cuántos filósofos pueden intentar comer a la vez (semáforo).
- Cambiar el orden de adquisición de tenedores.

- Asignar prioridades.

Otros problemas clásicos

- **Productor-consumidor:** compartir un buffer entre quienes producen y consumen datos.
- **Lectores-escriptores:** acceso concurrente a una base de datos.
 - Lectores pueden compartir, escritores deben excluir.

8. Comunicación entre Procesos (IPC)

Los procesos que cooperan necesitan compartir información o coordinarse.

Técnicas:

- **Memoria compartida:** muy rápida, pero requiere sincronización.
- **Pipes:** flujo unidireccional de datos.
- **Colas de mensajes:** estructuras organizadas para enviar datos.
- **Sockets:** para comunicación por red (TCP/UDP o local).

Usos actuales: pipelines de IA, microservicios, contenedores.

9. Interbloqueos (Deadlocks)

Un **interbloqueo** ocurre cuando varios procesos se quedan esperando recursos unos de otros, formando un ciclo y sin que ninguno pueda continuar.

Ejemplo: una rotonda donde cada coche bloquea al siguiente.

Condiciones de Coffman (deben cumplirse las cuatro):

1. **Exclusión mutua:** un recurso solo puede ser usado por un proceso.
2. **Retención y espera:** se mantiene un recurso mientras se espera otro.
3. **No expropiación:** no se pueden quitar recursos por la fuerza.
4. **Espera circular:** cadena cerrada de procesos esperando.

Cómo evitar o resolver:

- **Prevención:** se impide que se cumplan todas las condiciones. Ej.: pedir todos los recursos al inicio.
- **Evitación:** el sistema simula la asignación (algoritmo del banquero).
- **Detección y recuperación:** se permite que ocurra y luego se actúa (matar procesos, liberar recursos).

10. Comparativa por Sistema Operativo

SO	Gestión de procesos	Herramientas típicas

Linux/UNIX	<code>fork/exec</code> , <code>/proc</code> , CFS	<code>top</code> , <code>htop</code> , <code>strace</code>
Windows	<code>CreateProcess</code> , scheduler, multilevel	Task Manager, PowerShell
Windows Server	<code>Job objects</code> , prioridades extendidas	Hyper-V, Event Viewer

11. Procesos en Entornos Modernos

- **Contenedores:** usan *namespaces* para aislamiento y *cgroups* para controlar recursos.
 - Usan **namespaces** para que cada contenedor tenga su propio entorno aislado: su propia vista de procesos, red, sistema de archivos.
 - Usan **cgroups** para limitar el uso de recursos: CPU, memoria, red, disco.
- **IA:** entrenamiento en paralelo; procesos asignados a GPU separadas.
- **Bases de datos:**
 - PostgreSQL: modelo basado en procesos, cada proceso atiende un cliente
 - MongoDB: modelo multihilo, multiples hilos, multiples clientes
 - Redis: proceso único con event-loop optimizado.

12. Herramientas de Monitorización

- `ps`, `top`, `htop`: estado de procesos en tiempo real.
- `strace`: rastrea llamadas del sistema por proceso.
- `lsof`: muestra archivos abiertos por procesos.
- En Windows: `tasklist`, `taskkill`, PowerShell (`Get-Process`, `Stop-Process`).

13. Seguridad Informática en Procesos

Los procesos son potenciales vulnerabilidades si no se controlan:

Amenazas comunes:

- **Condiciones de carrera:** corrupción de datos, ataques.
- **Procesos zombie:** ya finalizados, pero no eliminados.
- **Señales maliciosas:** como `SIGKILL`, si no hay control de privilegios.
- **IPC insegura:** espionaje o manipulación de datos.

Protecciones:

- **ASLR:** aleatoriza direcciones de memoria.

- **Permisos UID/GID, SELinux, AppArmor.**
- **Contenedores/sandbox:** aíslan procesos.
- **Monitorización activa:** `auditd`, `fail2ban`, `Wazuh`.

Buenas prácticas:

- Ejecutar con privilegios mínimos (evitar `root` innecesario).
- Validar siempre la entrada.
- Usar sincronización segura

14. Conclusión

La gestión de procesos es el corazón de cualquier sistema operativo moderno. Nos permite construir sistemas **escalables, estables y seguros**, claves en la era del **cloud**, la **IA** y los **contenedores**.

Propuesta didáctica: "Simulador de Comedor de Filosofía"

Idea general

Diseñar un simulador interactivo en Python que modele el **problema de los cinco filósofos**, permitiendo al alumnado:

- Implementar procesos o hilos.
- Usar semáforos/mutex para sincronización.
- Experimentar con distintos enfoques de resolución del problema.
- Observar las consecuencias de interbloqueos, inanición o sincronización correcta.

Mini enunciado

Actividad: Filosofía en concurrencia

Cinco filósofos se sientan alrededor de una mesa. Cada uno alterna entre pensar y comer. Para comer necesita tomar los dos tenedores a sus lados.

Implementa este escenario con **procesos o hilos** y **mecanismos de sincronización (semáforos o mutex)**.

Crea al menos **dos versiones**:

- Una que produzca interbloqueo.
- Otra que lo evite correctamente. Muestra en consola los estados de los filósofos y evalúa el comportamiento observado.

Tema 20: Explotación y administración de sistemas operativos monousuario y multiusuario

1. Introducción

- Un **Sistema Operativo (SO)** es el software base que gestiona los recursos físicos y lógicos de un ordenador, permitiendo la interacción entre el usuario y el hardware.

Funciones principales:

- Gestión de procesos, memoria, almacenamiento, dispositivos, usuarios y redes.
- Interfaz entre aplicaciones y hardware.
- **Evolución:**
 - De sistemas monousuario y monotarea → a multitarea, multiusuario, distribuidos y en la nube.

2. Clasificación de los Sistemas Operativos

2.1. Según el número de procesadores

- **Monoprocesador:** ejecuta instrucciones en un solo núcleo (equipos antiguos).
- **Multiprocesador:** uso simultáneo de varias CPU o núcleos.
 - **SMP (Symmetric Multiprocessing):** todos los núcleos comparten memoria.
 - **NUMA (Non-Uniform Memory Access):** cada procesador accede a su propia memoria.

2.2. Según el número de usuarios

- **Monousuario (por uso práctico):**
 - Solo permite un usuario activo por sesión.
 - Ej.: Windows 11 Home, macOS (en uso doméstico).
- **Multiusuario (por capacidad técnica):**
 - Permiten múltiples sesiones concurrentes, locales o remotas.
 - Ej.: Linux (SSH), Windows Server, Unix, Solaris.

2.3. Según el número de tareas

- **Monotarea:** ejecuta solo una tarea a la vez (obsoleto).
- **Multitarea:** múltiples tareas en ejecución simultánea o concurrente.

2.4. Según la arquitectura del núcleo

- **Monolítico:** todo el SO reside en el espacio del kernel (Linux tradicional).
- **Microkernel:** servicios mínimos en el núcleo; resto, en espacio de usuario (Minix, QNX).
- **Híbrido:** combinación de ambos (Windows NT, macOS).

2.5. Según el entorno de ejecución

- **Sistemas en red:** comparten recursos entre equipos (Windows Server, FreeBSD).
- **Sistemas distribuidos:** múltiples máquinas operan como un solo sistema (Kubernetes, Apache Mesos).
- **Sistemas en la nube:** optimizados para infraestructura cloud (ChromeOS, AWS Lambda).

2.6. Según el tiempo de respuesta

- **Tiempo real (RTOS):** garantizan respuesta en un tiempo máximo determinado (VxWorks, QNX, FreeRTOS).

2.7. Sistemas operativos emergentes

- **IoT:** optimizados para bajo consumo y recursos limitados (Zephyr, RIOT OS).
- **Cuánticos:** controlan el acceso a qubits y algoritmos cuánticos (IBM Qiskit, Cirq).

3. Explotación de Sistemas Monousuario

3.1. Procedimientos habituales

- Instalación y configuración inicial del sistema.
- Gestión de cuentas de usuario (no simultáneas).
- Administración de software, actualizaciones, drivers y periféricos.

3.2. Niveles de explotación

- **Usuario:** acceso básico a programas y configuración del entorno.
- **Administrador:** gestión de recursos, seguridad, usuarios, copias de seguridad, etc.

3.3. Ejemplos actuales

- **Windows 11 Home/Pro:** interfaz gráfica, actualizaciones automáticas, configuración de privacidad.
- **macOS Sonoma:** gestión de perfiles de usuario, recursos compartidos, Time Machine, seguridad con FileVault.

4. Administración de Sistemas Multiusuario

4.1. Gestión de procesos

- Planificadores: **FIFO, Round Robin, prioridades, SJF, Multilevel Queue.**
- Comunicación entre procesos: **pipes, señales, sockets, colas de mensajes.**

4.2. Gestión de memoria

- Técnicas:
 - Memoria virtual.
 - Paginación, segmentación, swapping.
- Separación de espacios de direcciones: usuario / kernel.

4.3. Servicios del sistema

- **UNIX/Linux:**
 - **Daemons, systemd, crontab.**
- **Windows:**
 - **Servicios en segundo plano, Task Scheduler.**

4.4. Almacenamiento y sistemas de archivos

- **Sistemas: NTFS, EXT4, Btrfs, ZFS.**
- **Gestión de volúmenes lógicos: LVM, Storage Spaces.**

4.5. Administración avanzada

- **Linux Ubuntu Server:**
 - Gestión con **sudo**, ACLs, systemd, cgroups.
- **Windows Server 2022:**
 - Active Directory, control de acceso, GPOs, RDP.

5. Virtualización y Contenedores

5.1. Tipos de virtualización

- **Virtualización completa:** SO invitado sobre hardware virtual (VMware, VirtualBox, Hyper-V).
- **Paravirtualización:** usa parte del hardware del host (Xen, KVM).
- **Virtualización ligera:** contenedores con el mismo núcleo del host (Docker, LXC).

5.2. Contenedores y orquestación

- **Docker / Podman:** despliegue y gestión de contenedores.
- **Kubernetes:** orquestación de contenedores a escala.
- **Helm:** gestión de aplicaciones en Kubernetes.

5.3. Comparativa: VM vs. Contenedor

Característica	Máquina Virtual (VM)	Contenedor
Aislamiento	Alto (SO independiente)	Medio (comparten kernel)
Rendimiento	Menor	Mayor
Uso de recursos	Alto	Bajo
Tiempo de arranque	Lento	Rápido

6. Seguridad y Administración Avanzada

6.1. Seguridad

- **Control de acceso:** usuarios, permisos, ACLs, autenticación multifactor.
- **Cifrado:** BitLocker (Windows), LUKS (Linux), ZFS nativo.
- **Redes:** firewalls (iptables, nftables, Windows Defender).

6.2. Monitorización y rendimiento

- **Linux:** `htop`, `atop`, `vmstat`, Prometheus + Grafana.
- **Windows:** Monitor de rendimiento, Event Viewer, Sysinternals Suite.

6.3. Administración en entornos modernos

- **Cloud y Edge:**
 - IaaS: AWS EC2, Azure VMs.
 - PaaS: AWS Lambda, Azure Functions.
 - Edge: AWS Greengrass, Azure IoT Edge.

7. Tendencias y Futuro de los Sistemas Operativos

- **SO con Inteligencia Artificial:** adaptación al uso, predicción de tareas (Windows Copilot, AI Features en macOS).
- **Computación cuántica:** coordinación de operaciones cuánticas (Microsoft Quantum OS, IBM Q).
- **Serverless OS:** ejecución sin servidores gestionados (AWS Lambda, Google Cloud Run).
- **Sistemas auto-reparables y autónomos:** actualización en caliente, análisis predictivo.

8. Conclusión

- Los sistemas operativos han evolucionado hacia entornos **multiusuario, virtualizados y en la nube**.
- Su correcta administración implica dominar procesos, seguridad, contenedores y rendimiento.
- El futuro se centra en **eficiencia energética, automatización, IA y computación distribuida**.

Actividad: “Administra tu sistema: simulación de entornos monousuario y multiusuario”

Idea de la actividad:

El alumnado se organizará en parejas o pequeños grupos para **configurar, explotar y administrar un sistema operativo** en dos escenarios distintos: uno **monousuario** (como Windows 11 o macOS) y otro **multiusuario** (como Linux Ubuntu Server o Windows Server). El objetivo es comparar su estructura, funciones y modos de gestión desde una perspectiva práctica.

Tema 21: Sistemas informáticos. Estructura física y funcional

1. Introducción

- Definición general: combinación de hardware, software y redes para procesar, almacenar y transmitir información.
- Evolución: de sistemas centralizados (mainframes) a entornos distribuidos, inteligentes y sostenibles.
- Relevancia actual: base de la sociedad digital, desde el uso cotidiano hasta la industria avanzada.

2. Evolución, clasificación y tendencias

2.1. Evolución histórica

- Mainframes → PCs → Internet → Cloud → Edge → IA distribuida

2.2. Clasificación de los sistemas

- Por tamaño: microcomputadoras, servidores, supercomputadoras
- Por arquitectura: cliente-servidor, distribuido, embebido, cloud
- Por propósito: doméstico, empresarial, industrial, educativo, IoT

2.3. Tendencias actuales

- Serverless computing (ej. AWS Lambda), contenedores (Docker, Kubernetes)
- Edge/Fog Computing, virtualización ligera (MicroVMs)
- Sistemas heterogéneos (CPU+GPU+TPU+NPU), SoCs
- Computación cuántica y neuromórfica

3. Impacto social, económico y ético

- Transformación digital, industria 4.0, smart cities, sanidad conectada
- Brecha digital, accesibilidad, sostenibilidad (Green Computing)
- Legislación y ética:
 - Protección de datos (RGPD)
 - Soberanía digital
 - Transparencia algorítmica y sesgos
 - Computación responsable

4. Arquitectura física del sistema informático (Hardware)

4.1. Arquitectura básica

- Modelo Von Neumann: CPU, memoria, E/S, buses
- Alternativas: Harvard, SoC, arquitectura heterogénea

4.2. Procesamiento

- CPU: ALU, registros, control, caché, núcleos
- Coprocesadores: GPU, TPU, FPGA

4.3. Memoria y almacenamiento

- RAM (DDR5, HBM), memoria caché
- Almacenamiento: SSD (NVMe), HDD
- Cloud Storage: Amazon S3, Ceph, Google Cloud

4.4. Entrada/Salida y periféricos

- Tradicionales: teclado, ratón, pantalla, impresora
- Avanzados: sensores, IoT, VR/AR, biometría

4.5. Redes y conectividad

- Ethernet, Wi-Fi 6, 5G, SDN
- Edge Computing: procesamiento local distribuido

4.6. Energía y sostenibilidad

- Eficiencia energética, refrigeración líquida, renovables en CPDs

5. Arquitectura lógica y funcional (Software y sistema)

5.1. Capas funcionales

- Hardware → Firmware → Sistema Operativo → Middleware → Aplicaciones
- Modularidad y escalabilidad
- Interoperabilidad entre sistemas y plataformas

5.2. Software del sistema

- Sistemas operativos: Windows, Linux, Android, iOS
- Embebidos: RTOS, FreeRTOS, VxWorks

5.3. Software de aplicación

- ERP, CRM, diseño (CAD), multimedia, educativo, ofimática

5.4. Desarrollo y programación

- Lenguajes: Python, C++, Java, Go, Rust
- Entornos y herramientas: VS Code, IntelliJ, Git, CI/CD

5.5. Virtualización y contenedores

- Máquinas virtuales: VMware, KVM
- Contenedores: Docker, orquestadores (Kubernetes, OpenShift)
- MicroVMs: Firecracker, Kata Containers

5.6. Cloud-native software

- Aplicaciones distribuidas y escalables
- Microservicios, stateless apps, DevOps, SRE

6. Gestión de recursos y rendimiento

- Gestión de procesos y multitarea
- Planificación de CPU, asignación de memoria
- Técnicas: paginación, swapping, segmentación
- Monitorización: logs, métricas, herramientas (top, Grafana, Netdata)

7. Seguridad y protección

- Principios básicos: confidencialidad, integridad, disponibilidad
- Autenticación, autorización (OAuth2, Zero Trust)
- Amenazas comunes: phishing, malware, DDoS
- Medidas de protección: firewalls, backups, cifrado TLS 1.3

8. Aplicaciones y casos de uso reales

- **Medicina:** IA para diagnóstico, historia clínica electrónica
- **Educación:** aulas virtuales, plataformas de aprendizaje
- **Transporte:** vehículos autónomos, sensores inteligentes
- **Industria:** mantenimiento predictivo, robótica, IoT
- **Administración pública:** gestión digital, participación ciudadana

9. Conclusión

- Los sistemas informáticos son esenciales para la economía, la educación y la vida cotidiana.
- La tendencia es clara: más potencia, más inteligencia y más distribución.
- Desafíos actuales: ciberseguridad, eficiencia energética, ética digital.
- Futuro: integración con IA, computación cuántica, sistemas autónomos inteligentes.

Actividad: “Diseña la red y servicios de tu empresa”

Idea de la actividad:

El alumnado, organizado en grupos, diseñará la **infraestructura lógica y funcional de red** de una pequeña empresa ficticia (por ejemplo: una academia, una tienda, una gestoría o una clínica), simulando cómo se instalarían y configurarían los servicios en red necesarios para su funcionamiento.

Cada grupo deberá:

1. **Definir el escenario:** tipo de empresa, número de equipos, usuarios y necesidades básicas.
2. **Proponer una estructura de red** con:
 - Direccionamiento IP (privado).
 - Topología de red (cableado, switches, routers).
3. **Instalar y configurar servicios esenciales** (simulados o en máquina virtual):
 - Servidor DHCP y DNS.
 - Servidor de archivos (Samba/FTP).
 - Servidor web o de correo local.
 - Seguridad básica (firewall, control de acceso).
4. **Justificar las decisiones tomadas** desde el punto de vista técnico, funcional y organizativo.
5. **Presentar el diseño de forma visual y oral**, con esquemas de red, tabla de servicios y roles de los equipos.

Tema 22: Planificación y explotación de sistemas informáticos. Configuración. Condiciones de instalación. Medidas de seguridad. Procedimientos de uso.

1. Introducción

- Un sistema informático es una infraestructura tecnológica compuesta por hardware, software, redes y servicios.
- Su explotación eficiente requiere **planificación previa, instalación correcta, configuración segura y mantenimiento continuo**.
- Finalidad: asegurar **disponibilidad, escalabilidad, seguridad, rendimiento y cumplimiento normativo**.

2. Ciclo de vida de un sistema informático

Fases fundamentales:

1. **Análisis de necesidades**
2. **Diseño y planificación de la infraestructura**
3. **Adquisición e instalación**
4. **Configuración del sistema**
5. **Explotación, monitorización y mantenimiento**
6. **Actualización, migración o retirada**

3. Diseño y planificación del sistema

3.1. Estudio de necesidades

- Recursos: tipo de servicios, número de usuarios, disponibilidad requerida.
- Presupuesto, escalabilidad, tolerancia a fallos.

3.2. Elección de arquitectura

- **Monolítica, cliente-servidor, microservicios, cloud, edge.**
- Clasificación por modelo de despliegue: on-premise, cloud público/privado, híbrido, multicloud.

3.3. Selección de componentes

- Hardware: servidores, almacenamiento, red, energía.
- Software: sistema operativo, herramientas de administración, virtualización.

Ejemplo: elección de una solución cloud híbrida para una empresa con sedes distribuidas.

4. Instalación y condiciones técnicas

4.1. Instalación física y conectividad

- Centros de datos vs. edge computing.
- Cableado estructurado, Wi-Fi 6, redundancia eléctrica, climatización.

4.2. Seguridad física y ambiental

- UPS/SAI, control de acceso físico, sensores de temperatura/humedad.

4.3. Dispositivos y redes

- Equipamiento básico + periféricos + IoT.
- Protocolos de red modernos (IPv6, SD-WAN).

5. Configuración y despliegue

5.1. Gestión de la configuración

- Uso de CMDBs (bases de datos de configuración).
- Automatización con Ansible, Puppet, Chef.
- Versionado y despliegue controlado (GitOps).

5.2. Virtualización y contenedores

- Máquinas virtuales: VMware, KVM.
- Contenedores: Docker, Kubernetes.
- MicroVMs (Firecracker) para entornos ligeros.

Ejemplo: despliegue de un clúster de contenedores con Kubernetes en entorno educativo.

6. Explotación del sistema

6.1. Organización operativa

- Roles técnicos: administrador, operador, DevOps, SOC.

6.2. Automatización y DevOps

- CI/CD (Jenkins, GitLab), scripts de despliegue, pipelines.

6.3. Monitorización y mantenimiento

- Herramientas: Prometheus, Grafana, Zabbix, Netdata.
- Logs y alertas: ELK Stack, Graylog.

6.4. Alta disponibilidad y recuperación

- RAID, replicación, clústeres, backups automatizados.

Ejemplo: configuración de backup incremental diario y testeo de recuperación.

7. Seguridad del sistema

7.1. Seguridad general (CIA)

- Confidencialidad, integridad y disponibilidad como ejes.

7.2. Seguridad en red y dispositivos

- Firewalls (NGFW), IDS/IPS, VPN (WireGuard), control de acceso.

7.3. Seguridad en software

- Hardening (CIS Benchmarks), parches, control de versiones.

7.4. Cloud y contenedores

- Docker Bench, RBAC en Kubernetes, aislamiento de procesos.

7.5. Modelos avanzados

- Modelo Zero Trust, MFA, segmentación de red.

Ejemplo: implementación de políticas de privilegios mínimos + autenticación multifactor.

8. Procedimientos de uso y buenas prácticas

8.1. Normas operativas

- Políticas de contraseñas, backups, uso de dispositivos.

8.2. Gestión de usuarios y accesos

- LDAP, Active Directory, IAM, roles.

8.3. Formación y documentación

- Wikis internas (Confluence), manuales de uso, simulacros de seguridad.

8.4. Auditoría y trazabilidad

- Registro de eventos, SIEM (Wazuh, Splunk), cumplimiento normativo (ISO 27001, GDPR, ENS).

9. Conclusiones

- Los sistemas informáticos deben ser planificados con visión a largo plazo: seguros, flexibles y sostenibles.
- Tendencias clave: automatización, seguridad adaptativa, entornos híbridos.
- Retos: ciberseguridad, eficiencia energética, ética digital.

Actividad: “Proyecto técnico: configura y documenta tu sistema de red”

Idea de la actividad:

El alumnado, por parejas o grupos reducidos, diseñará y simulará el despliegue completo de un sistema informático para un caso realista (una pequeña empresa, un centro educativo, una tienda en red, etc.), abordando de forma integrada los aspectos de **planificación, instalación, configuración, seguridad y uso**, tal como se expone en el tema.

Cada grupo deberá:

1. **Diseñar el sistema:**
 - Elegir la arquitectura más adecuada (cliente-servidor, híbrida, etc.).
 - Establecer necesidades técnicas: número de usuarios, servicios requeridos, tipo de red.
2. **Simular la instalación:**
 - Proponer la ubicación del equipamiento (servidor, switches, puntos de acceso, UPS).
 - Diseñar la red física (cableado, direccionamiento IP, conectividad).
3. **Configurar los servicios:**
 - Configurar uno o varios servicios (por ejemplo, servidor web, DNS, DHCP, FTP o SAMBA).
 - Aplicar medidas básicas de seguridad: control de acceso, firewall, contraseñas seguras.
4. **Documentar y presentar:**
 - Elaborar una memoria técnica estructurada (con esquemas, tabla de configuración y justificaciones).
 - Presentar el diseño al resto de la clase, explicando las decisiones tomadas y las medidas de seguridad aplicadas.

Tema 23: Diseño de algoritmos. Técnicas descriptivas.

1. Introducción

- Un **algoritmo** es una secuencia finita de pasos definidos para resolver un problema.
- Fundamentales en programación, eficiencia computacional y resolución de tareas automatizadas.

1.1 Características principales

- **Precisión, determinismo, efectividad y fin de ejecución.**

1.2 Aplicaciones

- IA, Big Data, ciberseguridad, videojuegos, sistemas autónomos.

2. Elementos básicos de los algoritmos

2.1 Instrucciones fundamentales

- **Asignaciones, entrada/salida** y operaciones básicas.

2.2 Estructuras de control

- **Secuenciales**
- **Condicionales:** if, if-else, switch.
- **Iterativas:** for, while, repeat.

Mejora: evitar operaciones redundantes y validar entradas.

3. Representación de algoritmos

3.1 Pseudocódigo

- Lenguaje intermedio, legible y libre de sintaxis formal.
- Ideal para aprender a estructurar antes de codificar.

3.2 Diagramas de flujo

- Representación visual con flechas, rombos y óvalos.
- Muy usado en niveles iniciales y planificación visual.

3.3 Diagramas Nassi–Shneiderman (estructogramas)

- Bloques rectangulares alineados a estructuras de control.
- **Sin flechas**, lo que facilita la representación estructurada.
- Más compactos y directamente alineados con la programación estructurada moderna.

Comparativa	Diagrama de flujo	Nassi–Shneiderman
Visual	Flechas, formas	Bloques rectangulares
Flujo	Explícito	Implícito, por anidación
Ideal para	Comenzar a programar	Profundizar en estructura

Recomendados para FP, Bachillerato y programación estructurada.

3.4 Tablas de decisión

- Útiles para lógica compleja con múltiples reglas y salidas.

3.5 Diagramas Nassi-Shneiderman

- Bloques anidados que representan instrucciones, decisiones y bucles.
- Evitan saltos y facilitan una codificación directa.
- Mejoran la comprensión estructurada del algoritmo.

3.6 Entornos visuales y por bloques

- **Scratch, Snap!, Blockly, App Inventor.**
- Introducen la lógica algorítmica mediante bloques encajables.
- Adecuado para FP básica, ESO, y primeras fases del aprendizaje.

3.7 Asistentes inteligentes: LLMs y algoritmos

- Modelos como **ChatGPT, GitHub Copilot, Codex** pueden:
 - Convertir **instrucciones en lenguaje natural** a código.
 - Proporcionar ejemplos, detectar errores, optimizar soluciones.
- Promueven el aprendizaje guiado, accesible y rápido.

Ejemplo: “Haz un programa que calcule el factorial de un número” → Generación automática en Python, Java, etc.

3.8 Prototipado interactivo (Figma, Adobe XD)

- Herramientas como **Figma, Adobe XD** o **Penpot** permiten crear prototipos navegables de aplicaciones antes de codificar.
- Son útiles para representar de forma visual el **flujo algorítmico de pantallas** en interfaces de usuario.
- El alumno puede simular la lógica: *“Si el usuario pulsa en A, se va a la pantalla B”*.
- No requieren escribir código, pero sí pensar en **estructura, eventos y condiciones** (igual que un algoritmo visual).
- Ayudan a unir **algoritmia + diseño de interfaces** desde una perspectiva estructurada.

Ejemplo didáctico: diseñar en Figma el prototipo de una app de reservas donde las opciones y pantallas simulen el algoritmo planificado (if → pantalla X, else → pantalla Y).

4. Metodología de diseño algorítmico

4.1 Análisis del problema

- Entradas, salidas, restricciones.

4.2 Estructura de datos

- Arrays, listas, pilas, colas, grafos, árboles.

4.3 Diseño descendente (Top-Down)

- Dividir el problema en módulos o funciones más simples.

4.4 Verificación y pruebas

- Casos típicos y extremos. Evaluación de eficiencia.

5. Técnicas avanzadas

5.1 Divide y vencerás

- Separar en subproblemas → resolver → combinar.
- Ejemplo: Quicksort, Mergesort.

5.2 Programación dinámica

- Guardar resultados parciales (memorización).
- Ideal para problemas de optimización con subestructuras repetidas.

5.3 Algoritmos voraces (Greedy)

- Soluciones óptimas locales para aproximarse al global.
- Ejemplo: cambio de monedas, Dijkstra.

5.4 Backtracking

- Prueba y error sistemático con retroceso.
- Ejemplo: Sudoku, N reinas.

5.5 Algoritmos evolutivos

- Inspirados en biología (mutación, cruce).
- Aplicados en inteligencia artificial y optimización compleja.

6. Eficiencia algorítmica

6.1 Complejidad temporal y espacial

Notación	Ejemplo	Uso
$O(1)$	Acceso a array	Operación directa
$O(\log n)$	Búsqueda binaria	Árboles balanceados
$O(n)$	Recorrer lista	Lineal
$O(n \log n)$	Quicksort	Ordenaciones eficientes
$O(n^2)$	Bubble sort	Comparaciones simples
$O(n!)$	Permutaciones	Problemas combinatorios

7. Aplicaciones prácticas y casos reales

7.1 Reutilización de algoritmos

- Búsqueda, ordenación, recorridos.

7.2 Patrones de diseño

- Singleton, Strategy: aplicables a algoritmos modulares.

7.3 Casos destacados

- IA: backpropagation, clustering.
- Big Data: MapReduce.
- Ciberseguridad: RSA, cifrado hash.
- Computación cuántica: algoritmos de Grover y Shor.

8. Conclusiones y tendencias

8.1 Resumen comparativo

Técnica	Ventaja principal	Aplicación típica
Fuerza bruta	Fácil de implementar	Pruebas con pocos casos
Divide y vencerás	Escalable	Ordenación, búsqueda
Dinámica	Muy eficiente	Mochila, Fibonacci
Greedy	Rápido, aproximado	Cambios, caminos mínimos
Backtracking	Exploración completa	Juegos, combinaciones

8.2 Tendencias actuales y futuras

- Integración de IA para diseñar y optimizar algoritmos automáticamente.

- Avance de la **computación cuántica** y sus algoritmos asociados.
- Mayor accesibilidad con **programación visual y por bloques** para nuevos perfiles de programadores.

Idea de la actividad: “Del problema a la pantalla: diseña, representa y prototipa tu algoritmo”

Esta actividad propone que el alumnado afronte el **proceso completo de diseño algorítmico**, partiendo de un problema cotidiano expresado en lenguaje natural y recorriendo las distintas fases hasta llegar a una solución funcional y visualmente representada.

El objetivo es que el estudiante **piense algorítmicamente desde cero**, sin partir de código preestablecido, desarrollando su capacidad de abstracción, análisis, estructuración y traducción a distintos lenguajes: pseudocódigo, diagramas, lenguaje de programación y herramientas de diseño de interfaz.

Una vez desarrollado el algoritmo, el alumnado debe aplicar la lógica diseñada a un **prototipo visual interactivo** usando una herramienta como **Figma**, donde simulará la navegación entre pantallas o acciones, de forma coherente con la lógica definida previamente. Esto permite representar cómo se comportaría visualmente el sistema ante distintas decisiones o entradas del usuario, conectando así la lógica algorítmica con el diseño funcional de interfaces.

En resumen, la actividad permite **cerrar el ciclo de diseño computacional**, partiendo del análisis y finalizando en una representación interactiva, reforzando la conexión entre algoritmo, código y experiencia de usuario.

Tema 24: Lenguajes de programación: Tipos y características

1. Introducción

Un lenguaje de programación es un sistema formal que permite expresar algoritmos de forma precisa y comprensible para una máquina. Actúa como puente entre el pensamiento lógico del programador y la ejecución automatizada por parte del hardware.

Su evolución responde a las necesidades de cada época: eficiencia, mantenibilidad, seguridad, inteligencia artificial, desarrollo web, programación visual o computación cuántica.

2. Elementos básicos de un lenguaje de programación

2.1 Sintaxis y semántica

- **Sintaxis:** reglas de escritura del código (estructura, puntuación, orden).
- **Semántica:** significado lógico de las instrucciones escritas.

2.2 Estructuras de control

- **Secuenciales:** instrucciones que se ejecutan de forma lineal.
- **Condicionales:** `if`, `else`, `switch` → controlan el flujo según condiciones.
- **Iterativas:** `for`, `while`, `do while` → permiten repetir bloques.

2.3 Tipos de datos

- **Primitivos:** `int`, `char`, `float`, `bool` → valores simples.
- **Estructurados:** arrays, registros (`struct`) → agrupación de datos.
- **Abstractos:** listas, pilas, colas, árboles, grafos → modelan estructuras complejas.

2.4 Tipado

- **Estático** (Java, C++): el tipo se define en compilación.
- **Dinámico** (Python, JavaScript): se resuelve en ejecución.
- **Fuerte** (Python): no permite conversiones implícitas.
- **Débil** (JavaScript): sí permite conversiones automáticas.

2.5 Otras cualidades destacables

- **Legibilidad, modularidad, portabilidad, seguridad, eficiencia, mantenibilidad.**

3. Paradigmas de programación

Un paradigma es un modelo conceptual que guía cómo se estructura un programa. Algunos lenguajes permiten múltiples paradigmas (como Python o JavaScript).

A. Paradigmas tradicionales

Imperativo

Enfoque: indica *cómo* se hacen las cosas.

Ejemplo (C):

```
for(int i = 0; i < 10; i++) {  
  
    printf("%d\n", i);  
  
}
```

- **Declarativo**

Enfoque: expresa *qué* se quiere lograr, no cómo.

Ejemplo (SQL):

```
SELECT nombre FROM usuarios WHERE edad > 18;
```

- **Funcional**

Enfoque: funciones puras, sin efectos secundarios.

Ejemplo (Haskell):

```
sumaCuadrados x y = (x^2) + (y^2)
```

Lógico

Enfoque: define hechos y reglas, el sistema deduce soluciones.

Ejemplo (Prolog):

```
padre(juan, maria).
```

```
hermano(X, Y) :- padre(Z, X), padre(Z, Y), X \= Y.
```

B. Paradigmas orientados a estructuras

Orientado a objetos (OOP)

Enfoque: encapsula datos y comportamiento en objetos.

Ejemplo (Java):

```
public class Persona {  
  
    String nombre;  
  
    void saludar() {  
  
        System.out.println("Hola, soy " + nombre);  
  
    }  
  
}
```

- **Reactivo**

Enfoque: reacciona automáticamente a eventos o cambios.

Ejemplo (Vue.js): uso de `@click`, `v-if`, `watch`, `data`

Tiempo real

Enfoque: respuesta inmediata ante estímulos del entorno.

Ejemplo (C embebido):

```
while(1) {  
  
    if (temperatura() > 60) {  
  
        activarAlarma();  
  
    }  
  
}
```

```
}  
  
}
```

C. Paradigmas emergentes

Cuántico

Enfoque: basado en qubits, superposición y entrelazamiento.

Ejemplo (Q#):

```
operation HelloQuantum() : Unit {  
  
    using (qubit = Qubit()) {  
  
        H(qubit);  
  
        Message(";Hola desde un qubit en superposición!");  
  
        Reset(qubit);  
  
    }  
  
}
```

4. Clasificación de los lenguajes de programación

4.1 Por nivel de abstracción

- **Bajo nivel:** ensamblador → muy cercano al hardware.
- **Medio nivel:** C, Rust → equilibrio entre control y abstracción.
- **Alto nivel:** Python, Java → mayor facilidad para el programador.

4.2 Por forma de ejecución

- **Compilados:** C, C++
- **Interpretados:** Python, JavaScript
- **Híbridos:** Java, C#
- **Transpilados:** TypeScript → JavaScript

4.3 Por generación tecnológica

- **Clásicos:** Fortran, Pascal, COBOL
- **Modernos:** Kotlin, Go, Rust, Swift
- **Emergentes:** Q#, Cirq, Mojo (IA y cuántica)

5. Lenguajes y sus aplicaciones típicas

- **C/C++** → Sistemas operativos, drivers, sistemas embebidos
- **Java** → Aplicaciones empresariales y móviles (Android)
- **Python** → Inteligencia artificial, scripting, ciencia de datos, automatización
- **JavaScript / TypeScript** → Desarrollo web frontend y backend (Node.js)
- **Q#** → Programación cuántica (Azure Quantum)
- **SQL** → Gestión de bases de datos relacionales
- **Low-code / No-code** → Prototipado rápido y automatización visual (ej. Glide, Softr)

6. Herramientas y entornos de desarrollo

6.1 Procesadores de lenguajes

- **Compiladores:** gcc (C), javac (Java)
- **Intérpretes:** python, node
- **Ensambladores:** NASM, MASM

6.2 Entornos de desarrollo (IDE)

- **Locales:** Visual Studio Code, IntelliJ IDEA, Eclipse
- **En la nube:** GitHub Codespaces, Replit, CodeSandbox, Glitch

7. Tendencias actuales en programación

7.1 Cloud Computing

Desarrollo y escalado de apps en la nube (AWS, Azure). Lenguajes frecuentes: Python, Go, JavaScript.

7.2 Inteligencia Artificial y Machine Learning

Python como lenguaje dominante. Bibliotecas: TensorFlow, PyTorch, scikit-learn.

7.3 Programación visual: low-code / no-code

Creación de aplicaciones sin escribir código manual. Ejemplos: Softr, Glide, Appgyver.

7.4 Programación cuántica

Simuladores online (IBM Q Experience, Azure Quantum). Lenguajes: Q#, Cirq.

7.5 Trabajo colaborativo y remoto

Herramientas: GitHub Codespaces, VS Code Live Share, Replit.

8. Conclusión

- La elección del lenguaje depende del contexto del proyecto y del tipo de solución a implementar.
- Los distintos paradigmas conviven en la práctica profesional moderna.
- El conocimiento profundo de los fundamentos permite adaptarse a tecnologías emergentes.
- Estar al día con entornos cloud, IA y desarrollo colaborativo es clave para la empleabilidad.

Actividad: “Elige tu lenguaje: construye tu propio manifiesto de programación”

Idea de la actividad:

El alumnado, organizado en equipos, deberá **explorar, investigar, comparar y adoptar un lenguaje de programación** como si fueran una pequeña empresa tecnológica que va a elegir su lenguaje principal. A partir de esa elección, crearán un **manifiesto técnico** en el que justifiquen su decisión según características del lenguaje, su paradigma, sus

aplicaciones, herramientas asociadas, nivel de abstracción, entorno de ejecución y tendencias.

Pero lo más original es que deberán **"personificar" el lenguaje elegido**, defendiendo sus ventajas como si fuera un perfil profesional en una entrevista o pitch técnico. Al final, simularán un debate entre lenguajes en un "panel de selección" (como si compitieran por un puesto en una startup).

Actividad: "Diseña tu algoritmo estrella: el reality show de la eficiencia"

Idea de la actividad:

El alumnado, en grupos, se convertirá en un *equipo de desarrollo de software* que compite por diseñar el algoritmo más claro, eficiente y estructurado para resolver un problema cotidiano (por ejemplo: clasificar tareas, calcular rutas, controlar el acceso a un sistema, etc.). Cada equipo deberá aplicar estrictamente el paradigma de programación estructurada, utilizando funciones, estructuras de control y buenas prácticas de modularidad.

Fases de la actividad:

1. **Briefing del reto:** cada grupo recibe un problema informal (ej. "queremos organizar los turnos del comedor escolar" o "necesitamos una mini app de registro de asistencia").
2. **Diseño de solución estructurada:** deberán plantear su solución mediante:
 - Análisis del flujo (diagrama o pseudocódigo).
 - Uso justificado de secuencia, selección, iteración.
 - División en funciones/procedimientos con parámetros y retorno.
3. **"Casting de funciones" (Gamificación):**
Cada grupo deberá presentar una de sus funciones como si fuera un *candidato estrella*, explicando:
 - Su "rol" en el programa.
 - Qué hace, cómo se optimiza.
 - Por qué es eficiente y fácil de mantener.
4. **Simulación del "Código en acción":**
Mediante pseudocódigo o ejecución real (Python o pseudolenguaje), cada grupo demuestra su algoritmo funcionando.
5. **Rúbrica + panel de jueces (profesor + compañeros):**
Se evalúa según claridad, modularidad, eficiencia básica, presentación y uso de estructuras estructuradas.

Tema 25: Programación Estructurada. Estructuras Básicas. Funciones y Procedimientos.

1. Introducción a la Programación Estructurada

- Paradigma que organiza el código mediante estructuras de control bien definidas: secuencia, selección e iteración.
- Fomenta claridad, modularidad y mantenimiento eficiente del código.
- Punto de partida fundamental para aprender programación orientada a objetos, lógica o funcional.

1.1. Historia y contexto

- Surge en los años 70 con Dijkstra, como respuesta al desorden del "spaghetti code".
- Promueve la eliminación del uso indiscriminado de `goto` y fomenta estructuras con entrada/salida única.

1.2. Objetivos y beneficios

- Claridad, legibilidad y mantenimiento del código.
- Reducción de errores y facilidad de depuración.
- Reutilización de código mediante funciones/procedimientos.
- Facilita el trabajo colaborativo y las pruebas.

2. Estructuras de Control Básicas

2.1. Secuencia

- Ejecución ordenada de instrucciones, de arriba abajo.

```
a = 5
b = 10
print(a + b)
```

2.2. Selección (Condicionales)

- Permite decidir qué bloque de código ejecutar.
- If-Else, Switch-Case (o Match), operador ternario.

```
if edad >= 18:
    print("Adulto")
else:
    print("Menor")

estado = "OK" if respuesta else "Error"
```

2.3. Iteración (Bucles)

- Estructuras que repiten bloques: for, while, do-while.
- Uso de `break` y `continue` para controlar el flujo.

```
for i in range(5):  
    if i == 3:  
        continue  
  
    print(i)
```

3. Funciones y Procedimientos

3.1. Diferencias clave

- Funciones: devuelven un valor.
- Procedimientos: realizan acciones sin retorno.
- Ambos permiten dividir el código en módulos reutilizables.

3.2. Parámetros

- Por valor: se pasa una copia.
- Por referencia: se puede modificar el valor original.
- Por defecto: simplifican llamadas repetidas.

3.3. Ámbito de variables

- Variables locales (dentro de funciones), globales (accesibles globalmente).
- Variables estáticas (persisten entre llamadas).

3.4. Valores de retorno

- Pueden ser tipos simples o estructuras complejas.
- Usados para controlar el flujo y transmitir resultados.

3.5. Recursividad

- Una función se llama a sí misma.
- Requiere caso base y caso recursivo.

```
def factorial(n):  
    return 1 if n == 0 else n * factorial(n - 1)
```

3.6. Diseño modular

- División de programas en archivos y funciones independientes.
- Mejora la escalabilidad, pruebas y mantenimiento.

4. Control de Flujo Avanzado

4.1. Manejo de excepciones

- Diferencia entre error y excepción.
- Uso de try-except-finally.

```

try:
    resultado = 10 / divisor
except ZeroDivisionError:
    print("No dividir por cero")
finally:
    print("Fin")

```

4.2. Sentencias adicionales

- Goto: uso desaconsejado.
- Assert: verificación en tiempo de ejecución.

4.3. Excepciones personalizadas

- Creación de tipos propios de error para contextos específicos.

5. Optimización y Eficiencia

5.1. Análisis algorítmico

- Uso de notación Big-O para evaluar eficiencia.
- Comparación entre versiones iterativas y recursivas.

5.2. Gestión de recursos

- Uso eficiente de memoria y CPU.
- Evitar estructuras innecesarias o costosas.

5.3. Técnicas de optimización

- Memorización y caching para evitar recomputación.
- Refactorización estructurada para mantener claridad.

6. Comparación: Programación Estructurada vs. POO

Aspecto	Estructurada	Orientada a Objetos
Unidad principal	Funciones y procedimientos	Clases y objetos
Datos	Manipulación directa	Encapsulados y ocultos

Escalabilidad	Limitada a módulos	Alto nivel con herencia y polimorfismo
Ideal para	Scripts, herramientas, algoritmos	Sistemas complejos y reutilizables

7. Tendencias Modernas y Paradigmas Híbridos

7.1. Lenguajes modernos estructurados

- **Rust:** estructurado y seguro para sistemas.
- **Go:** sintaxis limpia, ideal para servicios concurrentes.
- **Python/TypeScript:** estructurados con tipado moderno.

7.2. Paradigmas híbridos

- Uso estructurado dentro de lenguajes OO.
- PE + programación funcional (Scala, Python, Kotlin).

7.3. Buenas prácticas estructuradas

- Funciones pequeñas y bien nombradas.
- Uso de herramientas modernas: linters, formateadores, Git.
- Documentación clara.
- Patrones estructurales como MVC o arquitectura por capas.

8. Casos Prácticos y Aplicaciones

8.1. Implementación de algoritmos

- Ordenamiento (QuickSort, MergeSort).
- Recorrido de estructuras: árboles, grafos.

8.2. Refactorización y optimización

- Detectar y corregir código ineficiente.
- Mejora incremental sin alterar funcionalidad.

8.3. Aplicación en estructuras de datos

- Pilas, colas, listas, árboles.
- PE permite manejar estructuras complejas de forma clara y predecible.

9. Conclusiones

- La programación estructurada sigue siendo esencial:
 - Es base pedagógica y técnica de otros paradigmas.
 - Se usa en sistemas embebidos, scripting, automatización.
- Su enfoque claro y modular mejora el desarrollo, la colaboración y la calidad del software.

Actividad: “Diseña tu algoritmo estrella: el reality show de la eficiencia”

Idea de la actividad:

El alumnado, en grupos, se convertirá en un *equipo de desarrollo de software* que compite por diseñar el algoritmo más claro, eficiente y estructurado para resolver un problema cotidiano (por ejemplo: clasificar tareas, calcular rutas, controlar el acceso a un sistema, etc.). Cada equipo deberá aplicar estrictamente el paradigma de programación estructurada, utilizando funciones, estructuras de control y buenas prácticas de modularidad.

Fases de la actividad:

1. **Briefing del reto:** cada grupo recibe un problema informal (ej. “queremos organizar los turnos del comedor escolar” o “necesitamos una mini app de registro de asistencia”).
2. **Diseño de solución estructurada:** deberán plantear su solución mediante:
 - Análisis del flujo (diagrama o pseudocódigo).
 - Uso justificado de secuencia, selección, iteración.
 - División en funciones/procedimientos con parámetros y retorno.
3. **“Casting de funciones” (Gamificación):**
Cada grupo deberá presentar una de sus funciones como si fuera un *candidato estrella*, explicando:
 - Su “rol” en el programa.
 - Qué hace, cómo se optimiza.
 - Por qué es eficiente y fácil de mantener.
4. **Simulación del “Código en acción”:**
Mediante pseudocódigo o ejecución real (Python o pseudolenguaje), cada grupo demuestra su algoritmo funcionando.
5. **Rúbrica + panel de jueces (profesor + compañeros):**
Se evalúa según claridad, modularidad, eficiencia básica, presentación y uso de estructuras estructuradas.

Tema 26: Programación modular. Diseño de funciones. Recursividad. Librerías.

1. Introducción

La programación modular surge como evolución natural de la programación estructurada, ante la necesidad de abordar proyectos de mayor escala, mantenibilidad y colaboración. Consiste en dividir un programa en **módulos independientes**, cada uno encargado de una tarea específica, que interactúan mediante **interfaces bien definidas**.

Objetivos:

- Mejorar la mantenibilidad y escalabilidad del código.
- Fomentar la reutilización y el trabajo colaborativo.
- Aislar errores y facilitar pruebas unitarias.

Anécdota técnica:

En una ocasión, olvidé definir el caso base en una función recursiva en C. Al ejecutarla con una entrada grande, el programa falló por desbordamiento de pila. Este error ilustra por qué es esencial una condición de parada clara para evitar bucles infinitos y saturación del sistema.

2. Fundamentos de la programación modular

2.1. ¿Qué es un módulo?

Un módulo es una unidad lógica y funcional de código que encapsula una tarea específica y se comunica con otros mediante interfaces públicas.

Características clave:

- **Alta cohesión:** todas las partes del módulo están relacionadas con un único propósito.
- **Bajo acoplamiento:** el módulo depende mínimamente de otros para funcionar.
- **Interfaz clara:** define de forma precisa qué datos recibe y qué resultados entrega.

Ejemplo (Python):

```
# modulo_usuario.py

def registrar_usuario(nombre):

    # lógica de registro

    return True
```

2.2. Cohesión y Acoplamiento

Cohesión: mide cuán bien agrupadas están las funcionalidades dentro del módulo.

- **Alta cohesión (positiva):** tareas estrechamente relacionadas. Ej: módulo de usuarios que solo gestiona altas, bajas y modificaciones.

- **Baja cohesión (negativa):** muchas tareas no relacionadas en un solo módulo. Ej: un módulo que gestiona usuarios y pagos.

Acoplamiento: mide el grado de dependencia entre módulos:

- **Acoplamiento bajo (deseado):** cada módulo puede cambiarse sin afectar a otros.
- **Acoplamiento alto (riesgoso):** cambios en un módulo afectan al resto.

Ventajas de alta cohesión y bajo acoplamiento:

- Reutilización de código.
- Mantenimiento más sencillo.
- Menor propagación de errores.

Inconvenientes si no se respetan:

- Dificultad para testear módulos aislados.
- Alta fragilidad ante cambios.
- Dependencias circulares y dificultad de escalado.

2.3. Circulación de datos entre módulos

- **Parámetros de entrada/salida** entre funciones.
- **Mensajes/eventos** en arquitecturas reactivas.
- **Interfaces públicas** que ocultan la implementación interna.

3. Diseño de funciones

3.1. Principios básicos

- Aplicar SRP: una función debe hacer una única tarea.
- Nombres claros y descriptivos.
- Limitar parámetros (idealmente ≤ 3); usar objetos si son más.

3.2. Variables globales vs. locales

- Las variables globales pueden producir efectos secundarios no deseados y dificultan el testeo.
- Buenas prácticas:
 - Preferir variables locales.
 - Encapsular el estado.
 - Usar gestores de estado (Vuex, Redux) en frontend.

3.3. Refactorización

- Dividir funciones extensas.
- Separar validación, lógica y presentación.
- Reutilizar subfunciones bien nombradas.

3.4. Estructura del proyecto

- Separar código en carpetas como `controllers/`, `services/`, `utils/`.
- Uso correcto de importaciones:

```
from utils.math import calcular_area
```

4. Recursividad

4.1. Estructura básica

Toda función recursiva debe tener:

- **Caso base:** condición que termina la recursividad.
- **Caso recursivo:** llamada a sí misma con un problema reducido.

```
def factorial(n):  
    return 1 if n == 0 else n * factorial(n - 1)
```

4.2. Optimización

- **Memorización:** almacenar resultados previos para evitar recomputación.
- **Tail recursion:** optimiza el uso de la pila si el lenguaje lo soporta.

4.3. Condiciones de parada y límites

- Toda función debe garantizar que se alcanzará el caso base.
- Recursividad profunda puede provocar stack overflow.
- Usar soluciones iterativas si el problema es simple o involucra grandes volúmenes.

4.4. Comparación recursiva vs. iterativa

Criterio	Recursivo	Iterativo
Claridad	Más expresivo para estructuras como árboles	Más simple para bucles planos
Rendimiento	Menor, por llamadas y pila	Mejor uso de CPU/memoria
Memoria	Mayor (uso de pila de llamadas)	Más eficiente
Casos ideales	Backtracking, árboles, problemas divididos	Conteo, acumulación, búsqueda lineal

5. Librerías y reutilización

5.1. Tipos

- **Librerías estándar:** integradas en el lenguaje (ej: `math`, `datetime` en Python).
- **Librerías externas:** instalables mediante gestores (`pip`, `npm`).

5.2. Buenas prácticas

- Documentar (`README.md`, `docstrings`).

- Versionado semántico (1.2.0).
- Evitar dependencias innecesarias.

5.3. Empaquetado

- Python: `setup.py`, `__init__.py`.
- JavaScript: módulos ES6, `package.json`.
- Java: `jar`, `package`.

6. Modularidad en sistemas reales

6.1. Microservicios

- Cada módulo se ejecuta como un servicio autónomo.
- Comunicación vía APIs REST o gRPC.

6.2. Arquitectura en capas

- División lógica: presentación, negocio, persistencia.
- Fomenta mantenimiento y pruebas separadas.

6.3. Comunicación entre módulos

- Interfaces REST.
- Mensajería (RabbitMQ, Kafka).
- Interoperabilidad mediante estándares abiertos.

6.4. Testing

- Permite pruebas unitarias aisladas.
- Frameworks: `pytest`, `unittest`, `Jest`.
- Facilita TDD y CI/CD.

7. Conclusión

La programación modular es un enfoque esencial para el desarrollo moderno.

Permite crear sistemas mantenibles, reutilizables y colaborativos. Su correcta aplicación, junto con el diseño eficiente de funciones, uso adecuado de recursividad y librerías bien gestionadas, constituye la base de cualquier arquitectura de software sólida y escalable.

Actividad: “¡Modula y vencerás! El concurso de arquitecturas limpias”

Idea de la actividad:

El alumnado, en equipos, asumirá el rol de una *consultora tecnológica* que compite para diseñar el sistema más **modular, mantenible y profesional**. Cada grupo deberá resolver un problema realista empleando principios de **programación modular**, diseñando funciones con buenas prácticas, gestionando dependencias y aplicando recursividad de forma justificada.

Fases de la actividad:

1. Misión por encargo (briefing del reto)

Cada grupo recibe una petición de cliente (problema informal), como por ejemplo:

- “Queremos un sistema modular de reservas para talleres escolares.”
- “Necesitamos un algoritmo para validar formularios de inscripción con condiciones personalizadas.”
- “Hay que diseñar una librería para gestionar productos y aplicar descuentos automáticamente.”

2. Diseño del sistema modular

Los equipos deben:

- Definir módulos funcionales y su propósito.
- Usar funciones bien diseñadas (parámetros, retorno, cohesión).
- Mostrar cómo circulan los datos (parámetros, estado compartido o eventos).
- Separar la lógica en archivos, carpetas o esquemas (según lenguaje elegido).

3. “La pasarela de funciones” (gamificación)

Cada grupo presenta su **función estrella**, como si fuera un “candidato ideal” a formar parte del sistema.

Deben explicar:

- Su función concreta dentro del módulo.
- Su diseño: claridad, entrada/salida, eficiencia.
- Cómo cumple con los principios de modularidad.

4. Recursividad inteligente

Si el problema lo permite, se valorará una función recursiva bien implementada.

Deben justificar:

- Por qué se ha usado recursividad.
- Cómo está asegurada la condición de parada.
- Si se ha optimizado (ej.: tail recursion, memorización).
O también explicar por qué prefieren una solución iterativa.

5. Demostración del sistema

- Presentan su estructura general (diagrama, pseudocódigo, ejecución).
- Explican cómo los módulos colaboran entre sí.
- Opcional: mostrar uso de librerías estándar o empaquetado del módulo.

6. Evaluación estilo “Comité de Arquitectura”

Se evalúa con rúbrica según:

- **Claridad estructural:** diseño modular, funciones bien definidas.
- **Buenas prácticas:** parámetros, nombres, refactorización.
- **Recursividad:** uso adecuado y seguro (si aplica).
- **Modularidad real:** separación lógica de responsabilidades.
- **Presentación:** defensa técnica y claridad de la explicación.

Tema 27: Programación orientada a objetos. Objetos. Clases. Herencia. Polimorfismo. Lenguajes.

1. Introducción y Motivación

- La programación orientada a objetos (POO) nace para modelar el mundo real en código, combinando **estado** (datos) y **comportamiento** (métodos) en unidades autocontenidas llamadas **objetos**.
- Frente a la rigidez de la programación estructurada, la POO permite construir sistemas **modulares, escalables y mantenibles**, alineados con la forma en que pensamos y colaboramos en equipo.

2. Evolución Histórica y Fundamentos

2.1. Origen

- **Programación estructurada (años 60–70):** C, Pascal → separación estricta entre funciones y datos.
- **Limitaciones:** dificultad para escalar, poca reutilización, rigidez del modelado.
- **Surgimiento de la POO (años 80):**
 - **Smalltalk:** primer lenguaje puramente orientado a objetos.
 - **C++ y Objective-C:** extensiones de C con POO.
 - **Consolidación:** Java, C#, Kotlin → POO moderna con seguridad y gestión automática.

2.2. Ventajas clave de la POO

- **Abstracción:** representación de entidades reales mediante clases.
- **Encapsulamiento:** protección del estado interno (`private`).
- **Herencia:** reutilización de comportamientos comunes.
- **Polimorfismo:** comportamiento flexible según el tipo real del objeto.

3. Fundamentos de la Programación Orientada a Objetos

3.1. Clases y Objetos

- **Clase:** molde general (ej.: `class Coche`).
- **Objeto:** instancia concreta (ej.: `miTesla = new Coche()`).

3.2. Atributos y Métodos

- **Atributos de instancia:** propios de cada objeto.
- **Estáticos:** compartidos (ej.: contador total).
- **Constantes:** valores fijos (`final` en Java).
- **Métodos de instancia:** actúan sobre el objeto (`acelerar()`).
- **Métodos estáticos:** no requieren objeto (`Coche.getTotal()`).

3.3. Visibilidad y Encapsulamiento

- **Modificadores:** `public`, `private`, `protected`.
- **Acceso controlado mediante getters/setters.**

```
private int velocidad;
public void setVelocidad(int v) { if (v >= 0) this.velocidad = v;
}
```

4. Relaciones entre Clases

4.1. Asociación, Agregación y Composición

- **Asociación:** uso sin dependencia fuerte (`Profesor usa Pizarra`).
- **Agregación:** relación no esencial (`Universidad → Estudiante`).
- **Composición:** dependencia total (`Casa → Habitaciones`).

4.2. Dependencias e Inyección

- Problema: clases muy acopladas (`Jugador` crea su `Arma`).
- Solución: **Inversión de Control** (IoC) y **Inyección de Dependencias** (DI).
- **Contenedores DI:** Spring, Unity → objetos inyectados automáticamente.

5. Pilares Principales de la POO

5.1. Abstracción e Interfaces

- **Clase abstracta vs. Interface:**
 - Clase abstracta → base con métodos comunes.
 - Interface → contrato de comportamiento.
- **Lenguajes:**
 - Java/C#: `interface`, `abstract class`.
 - Python: `ABC`, `@abstractmethod`.
- **Mixins:** herencia múltiple simulada (Python, Ruby).

5.2. Herencia

- **Tipos:** simple, multinivel, jerárquica, múltiple (C++).
- **Sobrescritura de métodos:** `@Override`.
- **Uso adecuado:** reutilizar comportamiento.
- **Abuso de herencia:** preferir composición cuando no hay relación "es-un".

5.3. Polimorfismo

- **Sobrecarga:** múltiples métodos con el mismo nombre y diferente firma.
- **Sobrescritura:** redefinir métodos en clases hijas.
- **Enlace dinámico:**

```
Animal a = new Perro(); a.hacerSonido(); // llama a Perro.ladRAR()
```

- **Paramétrico:** `List<T>` en Java/C#.

5.4. Principios SOLID

- **S:** una responsabilidad por clase.

- **O:** abierto a extensión, cerrado a modificación.
- **L:** sustitución segura de subclases.
- **I:** interfaces específicas, no generales.
- **D:** depender de abstracciones, no de implementaciones.

6. Lenguajes de Programación Orientados a Objetos

6.1. Comparativa

Lenguaje	POO pura	Herencia múltiple	Memoria	Usos principales
Java	Sí	No (interfaces)	GC	Web, Android
C++	Parcial	Sí	Manual	Juegos, sistemas
Python	Parcial	Sí (mixins)	GC	IA, scripting
C#	Sí	No (interfaces)	GC	.NET, escritorio

6.2. Casos de estudio

- **Java:** Spring (IoC, Beans, Repositorios).
- **C#:** Entity Framework (ORM + LINQ).
- **Python:** Django (modelos como clases, vistas como métodos).

7. Diseño Avanzado y Patrones

7.1. Patrones de diseño OO

Creacionales:

- Factory Method, Singleton, Abstract Factory.

Estructurales:

- Adapter, Decorator, Composite.
- Ejemplo: Decorator en Java para InputStream.

Comportamiento:

- Strategy, Observer, Command, State.

Ventajas:

- Reutilización, mantenibilidad, bajo acoplamiento.

7.2. Arquitecturas OO modernas

- **MVC:** separación en Modelo, Vista, Controlador.
- **Clean Architecture / Hexagonal:** dominio en el centro, independencia de frameworks.

- Aplicación en **microservicios**, APIs REST, backend reactivo.

8. Aplicaciones Prácticas y Buenas Prácticas

8.1. Ejercicios

- Jerarquía de figuras (clase abstracta **Figura**, clases **Círculo**, **Rectángulo**).
- Biblioteca (Libro, Revista con herencia).
- Sistema bancario: encapsulación + herencia + composición.

8.2. Buenas prácticas y errores comunes

- Evitar clases monolíticas.
- No abusar de herencia sin necesidad.
- Acoplamiento bajo mediante interfaces.
- Aplicar SOLID y refactorizar continuamente.

9. Tendencias Actuales y Conclusión

9.1. Tendencias

- **Hibridación con programación funcional:** Kotlin, Scala.
- **Orientación a objetos reactiva:** RxJava, ReactiveX.
- **Frameworks front/backend con POO:** Angular, React con TypeScript.
- **Contenedores y microservicios:** desacoplamiento total entre objetos distribuidos.

9.2. Conclusión

La POO es la base de la ingeniería de software moderna. Su enfoque modular, escalable y orientado al modelado del mundo real la convierte en esencial para el desarrollo profesional. Su combinación con paradigmas modernos permite construir sistemas robustos, mantenibles y adaptados al cambio.

Idea desarrollada: “La startup de los objetos: programa tu empresa”

Los alumnos se convierten en diseñadores de software en una startup ficticia que está a punto de lanzar su primer producto digital. Su misión no es solo programar, sino **modelar conceptualmente toda la lógica del sistema usando programación orientada a objetos (POO)**. Cada decisión de diseño deberá estar justificada como si formara parte de una arquitectura real de software: clara, mantenible, escalable y orientada al negocio.

La actividad comienza con la creación del universo de su startup: ¿a qué se dedica?, ¿quiénes son sus usuarios?, ¿qué procesos automatiza su software? A partir de esto, deben construir un ecosistema de clases que represente las entidades centrales del producto (por ejemplo, **Usuario**, **Producto**, **Pedido**, **Notificación**, **Pago**, etc.).

La gracia está en que no basta con listar clases: deberán aplicar **herencia**, **composición**, **encapsulamiento** y **polimorfismo de forma estratégica**, justificando su uso con argumentos técnicos. Además, deben pensar en el mantenimiento futuro del sistema, por lo que deberán incorporar **principios SOLID** y **al menos un patrón de diseño** que aumente la flexibilidad de su modelo.

Al finalizar, cada grupo presenta su “arquitectura OO” como si se tratara de una reunión técnica con inversores o stakeholders, explicando cómo su diseño resuelve los problemas

del dominio, permite escalar la aplicación y favorece el trabajo en equipo en entornos reales de desarrollo. La creatividad, la claridad en la comunicación y la aplicabilidad real del diseño serán clave.

Tema 31: Lenguaje C: Características generales. Elementos del lenguaje. Estructura de un programa. Funciones de librería y usuario. Entorno de compilación. Herramientas para la elaboración y depuración de programas en lenguaje C.

1. Introducción y Características Generales

1.1. Origen e impacto

- Desarrollado en 1972 por Dennis Ritchie (Bell Labs) como evolución de B.
- Reescribió UNIX y se consolidó como estándar en sistemas de bajo nivel.
- Base de lenguajes como C++, Java, Rust, Go y Objective-C.
- Estándares principales: ANSI C (C89), ISO C99, C11, C17.

1.2. Características clave

- **Medio nivel:** permite tanto abstracción como manipulación directa del hardware.
- **Portabilidad:** compila en casi cualquier sistema (Windows, Linux, microcontroladores).
- **Eficiencia extrema:** compilación directa a código máquina.
- **Sintaxis compacta:** solo 32 palabras clave en ANSI C.
- **Modularidad y estructura clara, aunque no soporta OOP nativamente.**

1.3. Ventajas y limitaciones

Ventajas	Limitaciones
Control directo del hardware	Gestión manual de memoria
Portabilidad y rendimiento	Tipado débil y sin seguridad por defecto
Gran documentación	Sin abstracciones modernas ni OOP
Ideal para sistemas embebidos	Curva de aprendizaje técnica

2. Elementos del Lenguaje

2.1. Estructura de un programa básico

```
#include <stdio.h>
```

```
int main() {  
    printf("Hola, mundo\n");  
    return 0;  
}
```

Componentes clave:

- Directivas (`#include`, `#define`)
- Función principal `int main()`
- Sentencias finalizadas con `;`
- Comentarios: `//` y `/* ... */`

2.2. Tipos de datos

- Primitivos: `char`, `int`, `float`, `double`
- Modificadores: `short`, `long`, `unsigned`, `signed`
- Cadenas como arrays de `char` terminadas en `\0`

2.3. Operadores

- Aritméticos: `+`, `-`, `*`, `/`, `%`
- Relacionales y lógicos: `==`, `!=`, `&&`, `||`
- Asignación y bit a bit: `=`, `+=`, `&`, `|`, `<<`

3. Estructura Modular de un Programa C

3.1. Separación por archivos

- `.h` → prototipos, constantes, estructuras
- `.c` → implementación

```
// operaciones.h
int sumar(int a, int b);
```

3.2. Preprocesador

- Macros (`#define`, `#ifdef`, `#ifndef`)
- Inclusión condicional:

```
#ifndef OPERACIONES_H
#define OPERACIONES_H
// ...
#endif
```

- Otros: `#pragma`, `#undef`, `#error`

4. Funciones: estándar y de usuario

4.1. Librería estándar

Cabecera	Funciones destacadas
<code><stdio.h></code>	<code>printf()</code> , <code>scanf()</code> , <code>fopen()</code>
<code><stdlib.h></code>	<code>malloc()</code> , <code>free()</code> , <code>atoi()</code>

<code><string.h></code>	<code>strlen(), strcpy(), strcmp()</code>
<code><math.h></code>	<code>sqrt(), pow(), fabs()</code>

4.2. Funciones definidas por el usuario

```
int suma(int a, int b) {
    return a + b;
}
```

- Declaración en `.h`, definición en `.c`
- **Parámetros por valor y referencia (punteros)**
- **Recursividad:**

```
int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}
```

5. Punteros y Gestión de Memoria

5.1. Declaración y uso de punteros

Los punteros son variables que almacenan direcciones de memoria, lo que permite manipular directamente valores ubicados en distintas partes del programa.

```
int a = 5;           // Variable normal
int *p = &a;         // p apunta a la dirección de a
```

- `*p` permite acceder al valor almacenado en la dirección (desreferenciación).
- `&a` obtiene la dirección de la variable `a`.

Importancia: permite paso por referencia, manejo eficiente de arrays y estructuras dinámicas, interacción con APIs de bajo nivel y sistemas embebidos.

5.2. Memoria dinámica

Se usa cuando no se conoce de antemano el tamaño de los datos, como arrays variables o estructuras complejas.

```
int *arr = malloc(10 * sizeof(int)); // reserva espacio para 10
enteros
// uso de arr[0], arr[1], ...
free(arr); // libera memoria cuando ya no se usa
```

Funciones comunes:

- `malloc(size)`: reserva memoria sin inicializar.
- `calloc(n, size)`: reserva e inicializa a cero.
- `realloc(ptr, new_size)`: cambia el tamaño de una zona reservada.
- `free(ptr)`: libera la memoria reservada.

5.3. Problemas comunes al usar punteros

- **Memory leaks:** no liberar memoria con `free()` → consumo progresivo de RAM.
- **Punteros no inicializados:** acceso a direcciones inválidas → comportamiento indefinido.
- **Buffer overflow:** escribir fuera de los límites de un array → errores graves, vulnerabilidades.
- **Segmentation fault:** acceso ilegal a memoria protegida → programa se detiene bruscamente.

Consejo: inicializar siempre punteros, verificar `malloc` y usar herramientas como Valgrind para detección.

6. Entorno de Compilación

6.1. Fases del proceso de compilación

1. **Preprocesado (.i):** Expande macros y directivas (`#include`, `#define`).
2. **Compilación (.o):** Traduce C a código máquina intermedio.
3. **Enlazado (linking):** Une múltiples archivos objeto y bibliotecas en un ejecutable.
4. **Ejecución:** El binario se ejecuta en el sistema operativo.

6.2. Herramientas del entorno

GCC: compilador más utilizado en Unix/Linux.

```
gcc -Wall -o programa archivo.c
```

- `-Wall`: activa todas las advertencias.
- `-o`: nombre del ejecutable.

Clang: compilador alternativo (basado en LLVM), destaca por su velocidad y mensajes claros.

Make y Makefile avanzado: automatiza la compilación de proyectos con múltiples archivos

```
CC = gcc
CFLAGS = -Wall -O2
OBJ = main.o operaciones.o
programa: $(OBJ)
    $(CC) $(CFLAGS) -o programa $(OBJ)
clean:
    rm -f *.o programa
```

- `clean`: elimina archivos intermedios.
- `-O2`: optimización de compilación.

7. Depuración, Testing y Optimización

7.1. Depuración con GDB (GNU Debugger)

Permite ejecutar el programa paso a paso y observar su comportamiento en tiempo real.

```
gdb ./programa
(gdb) break main      // punto de ruptura
(gdb) run             // ejecuta el programa
(gdb) next            // ejecuta línea a línea
(gdb) print variable  // inspecciona el valor de una variable
```

Ideal para encontrar errores lógicos y de punteros.

7.2. Validación de errores

Valgrind: detecta errores de memoria (fugas, uso de memoria no inicializada, escritura fuera de límites).

```
valgrind ./programa
```

Sanitizers (de GCC/Clang): detectan errores en tiempo de ejecución.

```
gcc -fsanitize=address -g archivo.c
```

- `-g`: añade símbolos de depuración.
- `-fsanitize=address`: detecta accesos inválidos a memoria.

7.3. Testing estructurado

Verifica que el programa cumpla con lo esperado.

- **Asserts:**

```
#include <assert.h>
assert(suma(2, 2) == 4);
```

- **Frameworks de testing en C:**
 - **Check** (popular en entornos POSIX).
 - **Unity** (ligero, ideal para sistemas embebidos).

7.4. Optimización y profiling

Niveles de optimización del compilador:

- `-O0`: sin optimizar (útil para depuración).
- `-O1`, `-O2`, `-O3`: niveles progresivos de optimización.
- `-Os`: optimización para espacio.

Gprof: herramienta para análisis del rendimiento.

```
gcc -pg archivo.c -o programa
./programa
gprof programa gmon.out
```

Genera un perfil de funciones más costosas y cuellos de botella.

8. Proyecto Integrado: Programa Modular en C

8.1. Estructura

```
mi_proyecto/  
├── main.c  
├── operaciones.c  
├── operaciones.h  
└── Makefile
```

8.2. Código y compilación

- Modulación clara entre declaraciones y lógica
- Uso de `make` para compilar y limpiar

9. Comparativa: C frente a otros lenguajes

Lenguaje	Ventajas frente a C	Inconvenientes frente a C
Python	Más simple y legible	Más lento, no control de memoria
Java	Orientado a objetos, seguro	Requiere máquina virtual
Rust	Memoria segura, sin GC	Mayor complejidad inicial

10. Conclusión y Valor Didáctico

- C sigue siendo fundamental en programación de sistemas, embebidos, compiladores y control de hardware.
- Es la mejor puerta de entrada para **entender la memoria, el rendimiento y la arquitectura del software a bajo nivel**.
- Su aprendizaje fortalece las competencias lógicas, algorítmicas y estructurales del alumnado.

Idea de actividad educativa: “C-Reto: Construye tu microkernel modular en equipo”

Concepto:

El alumnado, dividido en pequeños equipos de desarrollo, deberá crear un **programa modular en lenguaje C** que simule el funcionamiento básico de un sistema operativo o dispositivo embebido (por ejemplo, gestión de tareas, control de sensores, simulación de usuarios o periféricos). La clave será **estructurar el código de forma profesional**, con archivos `.h`, `.c`, uso de Makefiles, funciones reutilizables y validación mediante `assert` o `Valgrind`.

Desarrollo:

1. **Fase de diseño:**
Cada grupo define los módulos del sistema (entrada/salida, lógica, datos, depuración, etc.) y su interfaz (`.h`).
2. **Fase de implementación:**
Escriben el código modular usando funciones de usuario y algunas estándar (`stdio`, `stdlib`, etc.), aplicando punteros, control de memoria dinámica y uso del preprocesador.

3. **Fase de prueba y depuración:**

Usan `GDB`, `Valgrind`, `assert`, e introducen deliberadamente errores para que otros grupos los localicen ("modo cazador de bugs").

4. **Fase final:**

Compilan todo con un `Makefile` avanzado y presentan su solución, explicando las decisiones técnicas tomadas y cómo su diseño modular ayuda al mantenimiento y escalabilidad del sistema.

Objetivos educativos:

- Consolidar el uso del lenguaje C en un contexto realista y colaborativo.
- Practicar el diseño modular, la depuración y la gestión de errores.
- Fomentar la colaboración en programación profesional.
- Aprender a usar herramientas de desarrollo y testing.

Tema 32: Lenguaje C. Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.

1. Introducción

- C es un lenguaje clave en la programación de sistemas debido a su control total sobre memoria, estructuras de datos y bajo nivel.
- Este tema aborda cómo C permite construir estructuras estáticas y dinámicas, controlar la memoria manualmente, modularizar programas y aplicar punteros avanzados.

2. Estructuras de Datos Estáticas

2.1. Arrays

- Homogéneos, tamaño fijo en tiempo de compilación.
- Acceso por índice rápido (`arr[i]`), pero sin funciones de redimensionamiento.

2.2. `struct`

- Agrupación de datos heterogéneos.

```
struct Alumno { char nombre[50]; int edad; float nota; };
```

2.3. `union`

- Todos los campos comparten espacio en memoria

```
union Dato { int i; float f; char c; };
```

2.4. `enum`

- Definición de constantes simbólicas.

```
enum Estado { ACTIVO, INACTIVO, BORRADO };
```

3. Estructuras Dinámicas en C

3.1. Memoria dinámica

- `malloc`, `calloc`, `realloc`, `free`.
- Se debe verificar el resultado y liberar manualmente.

```
int* v = malloc(10 * sizeof(int));  
if (v) { /* usar */ }  
free(v);
```

3.2. Listas enlazadas

```

struct Nodo {
    int dato;
    struct Nodo* sig;
};

void insertar(struct Nodo** cabeza, int valor) {
    struct Nodo* nuevo = malloc(sizeof(struct Nodo));
    nuevo->dato = valor;
    nuevo->sig = *cabeza;
    *cabeza = nuevo;
}

```

3.3. Árbol binario de búsqueda (ABB)

```

struct Nodo {
    int dato;
    struct Nodo* izq;
    struct Nodo* der;
};

void insertar(struct Nodo** r, int v) {
    if (*r == NULL) {
        *r = malloc(sizeof(struct Nodo));
        (*r)->dato = v; (*r)->izq = (*r)->der = NULL;
    } else if (v < (*r)->dato)
        insertar(&(*r)->izq, v);
    else
        insertar(&(*r)->der, v);
}

```

3.4. Pilas y Colas

- Pila: LIFO. Implementación con **push** y **pop**.
- Cola: FIFO. Requiere punteros a cabeza y cola.

3.5. Tabla Hash

- Array de listas enlazadas (colisiones) o direccionamiento abierto.
Función hash: **clave % tamaño**.

3.6. Grafos

- Representación por listas de adyacencia (arrays de punteros a listas).

```

struct NodoAdy {
    int destino;
    struct NodoAdy* sig;
};

```

4. Entrada y Salida

4.1. Entrada estándar

- `scanf`, `fgets`, `getchar`.
- `fgets` es más segura para strings.

4.2. Archivos

```
FILE* f = fopen("datos.txt", "r");
if (f) { fgets(buffer, 100, f); fclose(f); }
```

5. Punteros y Gestión de Memoria

5.1. Fundamentos

- `&`: dirección. `*`: desreferencia.
- Punteros permiten modificar variables desde funciones.

5.2. Paso por referencia

```
void cambiar(int* p) { *p = 42; }
```

5.3. Punteros dobles

- Modifican punteros en funciones, matrices dinámicas, listas doblemente enlazadas.

6. Punteros a Funciones

6.1. Declaración

```
int suma(int a, int b) { return a + b; }
int (*pf)(int, int) = suma;
```

6.2. Menús dinámicos

```
void ver() { ... }
void editar() { ... }
```

```
void (*menu[])(void) = {ver, editar};
menu[opcion]();
```

6.3. Callbacks

- Ej: `qsort`, simuladores de eventos, intérpretes de comandos.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int comparar(const void* a, const void* b) {
    int x = *(int*)a;
    int y = *(int*)b;
    return x - y;
}
```

```
int main() {
    int datos[] = {5, 2, 9, 1, 6};
    int n = sizeof(datos) / sizeof(datos[0]);
```

```

    qsort(datos, n, sizeof(int), comparar);
    for (int i = 0; i < n; i++) {
        printf("%d ", datos[i]);
    }
    return 0;
}

```

Explicación: qsort ordena un array genérico. El cuarto parámetro es un puntero a función que define cómo comparar los elementos. Aquí, comparar los trata como int.

7. Modularización y Proyecto Integrado

7.1. Archivos

- `.h` para declaraciones, `.c` para implementación.
- `Makefile` para automatización:

```

CC=gcc
OBJ=main.o lista.o
all: prog
prog: $(OBJ)
    $(CC) -o prog $(OBJ)
clean:
    rm -f *.o prog

```

7.2. Proyecto

- Lista enlazada con carga/guardado en archivo.
- Menú interactivo con punteros a funciones.

8. Herramientas y Buenas Prácticas

8.1. Depuración

- Valgrind: fugas, uso de memoria inválida.
- GDB: inspección de ejecución paso a paso.

8.2. Testing

8.2. Testing

- `assert` permite validaciones simples y rápidas en tiempo de ejecución:

```

#include <assert.h>

assert(suma(2, 2) == 4);

```

- **CMocka**: framework moderno de testing para C, ligero y compatible con proyectos reales. Permite separar los tests del código principal y genera informes automáticos.

Ejemplo básico con CMocka: test de función suma()

```

// funciones.c

```

```

int suma(int a, int b) {
    return a + b;
}

// test_funciones.c

#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "funciones.h"

static void test_suma(void **state) {
    assert_int_equal(suma(2, 2), 4);
    assert_int_equal(suma(-1, 1), 0);
}

int main(void) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(test_suma),
    };

    return cmocka_run_group_tests(tests, NULL, NULL);
}

```

Compilación con CMocka (usando pkg-config):

```

gcc funciones.c test_funciones.c -o test -lcmocka -I/usr/include
-L/usr/lib

./test

```

Ventajas: ejecución automatizada, separación de código de prueba, posibilidad de mocks y test de errores en punteros.

8.3. Buenas prácticas

- Inicializar punteros.
- Verificar `malloc`.
- Documentar qué módulo reserva/libera memoria.
- Evitar punteros colgantes (`dangling pointers`).

- Separar lógica, entrada/salida y validación.

Idea de actividad didáctica: “Misión: estructura viva – modela, enlaza y ejecuta”

Concepto:

Los estudiantes se convierten en un equipo de programadores que recibe una misión: diseñar, implementar y probar **una estructura de datos dinámica en lenguaje C** para gestionar un sistema real (ej.: historial de navegación, gestor de tareas, control de accesos, cola de impresión). El reto consiste en combinar **punteros, estructuras, memoria dinámica, entrada/salida y punteros a funciones**, integrando todos los bloques del tema.

Etapas de la actividad:

1. Diseño conceptual

El grupo elige el sistema a simular y decide qué estructura necesita: ¿lista? ¿pila? ¿cola circular? ¿una combinación? Plasman su diseño en papel o diagrama de flujo/UML.

2. Implementación modular en C

Programan el sistema usando `struct`, `malloc/free`, punteros dobles y E/S por consola. El código se divide en funciones claras y se usa `Makefile`.

Punteros a funciones

Implementan un menú dinámico donde las opciones invocan funciones mediante punteros:

```
void (*acciones[])(void) = {insertar, eliminar, mostrar};
```

3. Validación y testeo

Usan `assert`, `Valgrind` y si pueden, `GDB` para probar su código. Documentan errores y mejoras.

4. Presentación tipo demo técnica

Cada equipo expone su diseño, muestra la ejecución real, y explica cómo gestionaron la memoria y modularidad.

Tema 36: La manipulación de datos. Operaciones. Lenguajes. Optimización de consultas.

1. Introducción

- Manipular datos es el núcleo funcional de todo SGBD: almacenar, recuperar, modificar y eliminar información de forma eficiente y segura.
- Tiene aplicación directa en aplicaciones web, móviles, IoT, sistemas de control y plataformas de Big Data.
- El rendimiento de las aplicaciones depende en gran medida del diseño de consultas y del modelo de datos.

2. Modelos de Datos

2.1. Modelo Relacional (SQL)

- Basado en **tablas**, relaciones entre entidades mediante claves primarias y foráneas. Utiliza la **normalización** para evitar redundancias y asegurar consistencia.

```
CREATE TABLE clientes (  
  id INT PRIMARY KEY,  
  nombre VARCHAR(100),  
  email VARCHAR(100)  
);
```

2.2. Modelos NoSQL

Documental: MongoDB guarda estructuras JSON anidadas.

```
{ "cliente": "Lucía", "pedidos": [{ "producto": "libro", "cantidad":  
2 }] }
```

- **Clave-valor:** Redis guarda pares directos, útil para caché.
- **Columnares:** Cassandra guarda columnas agrupadas por familias.
- **Grafos:** Neo4j representa nodos y aristas para relaciones complejas como redes sociales.

2.3. Modelos híbridos (NewSQL y multimodelo)

- ArangoDB: combina documentos, grafos y relaciones.
- CockroachDB: base relacional distribuida con propiedades ACID.

3. Lenguajes de Manipulación de Datos

3.1. SQL

DML (Data Manipulation Language):

```
SELECT nombre FROM clientes WHERE ciudad = 'Madrid';  
INSERT INTO pedidos (id, cliente_id, total) VALUES (1, 2, 45.6)
```

DDL (Data Definition Language): define estructura.

```
CREATE TABLE productos (...);
```

DCL (Data Control Language):

```
GRANT SELECT ON clientes TO 'usuario_lectura';
```

TCL (Transaction Control Language):

```
BEGIN; UPDATE ...; COMMIT;
```

3.2. NoSQL

MongoDB (MQL):

```
db.clientes.find({ ciudad: "Madrid" });
```

Cassandra (CQL):

```
SELECT * FROM clientes WHERE id = 3;
```

3.3. SQL moderno con JSON

PostgreSQL permite consultas a campos JSON:

```
SELECT datos->>'nombre' FROM empleados WHERE datos->>'pais' = 'España';
```

4. Operaciones de Manipulación de Datos

4.1. Operaciones básicas

- **SELECT**: recuperación.
- **INSERT, UPDATE, DELETE**.

```
DELETE FROM pedidos WHERE total = 0;
```

4.2. Consultas avanzadas

JOIN: combinar tablas relacionadas.

```
SELECT c.nombre, p.total  
FROM clientes c  
JOIN pedidos p ON c.id = p.cliente_id;
```

Subconsultas:

```
SELECT nombre FROM productos WHERE precio > (SELECT AVG(precio) FROM productos);
```

Agregación:

```
SELECT ciudad, COUNT(*) FROM clientes GROUP BY ciudad;
```

4.3. Transacciones

- Propiedades **ACID**:
 - Atomicidad, Consistencia, Aislamiento, Durabilidad.
- Control de concurrencia con niveles de aislamiento y **MVCC** (multi-version concurrency control).

5. Manipulación en NoSQL

5.1. MongoDB

Inserción:

```
db.clientes.insertOne({ nombre: "Eva", ciudad: "Madrid" });
```

Agregación:

```
db.pedidos.aggregate([
  { $match: { total: { $gt: 50 } } },
  { $group: { _id: "$cliente", total: { $sum: "$total" } } }
])
```

5.2. Cassandra

- Alta velocidad, pero sin **JOIN** o subconsultas.
- Consultas optimizadas para claves de partición.

6. Optimización de Consultas

6.1. En SQL

6.1.1. Índices

- B-Tree, Hash, GIN (para texto completo).

```
CREATE INDEX idx_ciudad ON clientes(ciudad);
```

6.1.2. Planificación

EXPLAIN muestra el plan de ejecución:

```
EXPLAIN ANALYZE SELECT * FROM pedidos WHERE cliente_id = 5;
```

6.1.3. Reescritura de consultas

- Evitar subconsultas innecesarias.
- Reemplazar **SELECT *** por columnas específicas.
- Usar **JOIN** correctos en vez de condiciones incorrectas.

⚠ Caso de error frecuente

<https://blogfbd.blogspot.com/2015/04/servidor-mysql-denuncia-sus-usuarios.html>

Consulta incorrecta:

```
SELECT COUNT(*) FROM articulo, memoria WHERE articulo.cod !=
memoria.cod;
```

- Provoca **producto cartesiano** con resultados absurdos.
-  Solución:

```
SELECT COUNT(*) FROM articulo WHERE cod NOT IN (SELECT cod FROM
memoria);
```

6.2. En NoSQL

- Índices en MongoDB (**TTL**, compuestos).
- Denormalización controlada.

- Diseño orientado a la consulta esperada (“query-first”).

6.3. Herramientas

- PostgreSQL: `pg_stat_statements`, `auto_explain`.
- MongoDB: `explain()`, MongoDB Atlas.
- Cassandra: `nodetool`, `tracing`.

7. Configuración y Escalabilidad

7.1. Parámetros configurables

- Tamaño de buffers, cachés, límite de conexiones.

7.2. Alta disponibilidad

- SQL: clustering, replicación master-slave.
- NoSQL: sharding, réplicas, tolerancia a particiones.

7.3. Entornos cloud

- Bases como MongoDB Atlas o AWS RDS permiten escalado automático y monitorización.

8. Tendencias y Futuro

8.1. Inteligencia Artificial

- Bases de datos autogestionadas (Oracle Autonomous DB).

8.2. Multimodelo

- Motores como OrientDB o ArangoDB integran modelos relacional, documental y grafo.

8.3. Big Data

- Integración con Spark, Hadoop, Kafka.
- Bases columnar como ClickHouse para análisis masivo.

9. Conclusión

- Manipular datos con rigor y eficiencia es esencial para la salud de cualquier sistema.
- Conocer el modelo, el lenguaje y la lógica de consultas permite prevenir errores críticos de rendimiento.
- SQL y NoSQL no son opuestos: se complementan en sistemas modernos y escalables.

Actividad: “Diseña tu Base de Datos del Mundo Real”

Idea General

El alumnado se convierte en un equipo de arquitectos de datos para una aplicación real (red social, tienda online, app de reservas, etc.). Su objetivo es diseñar desde cero una base de datos funcional, simular operaciones reales y optimizar consultas según escenarios.

Tema 39: Lenguajes para la definición y manipulación de datos en sistemas de Bases de Datos Relacionales. Tipos. Características. Lenguaje SQL

1. Introducción

- **SQL** es el lenguaje estándar para trabajar con bases de datos relacionales. Fue creado en los años 70 pero sigue siendo esencial.
- No es un lenguaje de programación general, sino **declarativo**: describe qué queremos obtener, no cómo hacerlo.
- Es adaptable a nuevas tecnologías (cloud, IA, Big Data) gracias a su solidez y evolución.
- Se integra fácilmente con otros lenguajes y plataformas (Python, JavaScript, R, Spark, etc.).

Piensa en SQL como el “idioma universal” para hablar con bases de datos estructuradas.

2. Componentes del lenguaje SQL

Aquí vemos cómo se divide SQL en sublenguajes, cada uno con su función:

Subtipo	¿Para qué sirve?	Ejemplos
DDL	Crear y modificar estructuras	CREATE TABLE, ALTER TABLE
DML	Manipular datos	INSERT, UPDATE, DELETE
DQL	Consultar información	SELECT
DCL	Controlar permisos	GRANT, REVOKE
TCL	Gestionar transacciones	BEGIN, COMMIT, ROLLBACK

Conocer esta clasificación te permite identificar qué bloque estudiar según el tipo de tarea que debas realizar.

3. DDL – Lenguaje de Definición


- Crea y organiza la “forma” de la base de datos (tablas, vistas, tipos).

Ejemplo:

```
CREATE TABLE empleados (  
  id INT PRIMARY KEY,
```

```
nombre VARCHAR(100),  
salario DECIMAL(8,2)  
);
```

- Tipos modernos como **JSON** o **GEOMETRY** permiten guardar datos complejos.
- Las **vistas materializadas** son muy útiles para informes que no cambian constantemente.
- **Triggers** permiten que la base de datos reaccione automáticamente a ciertos eventos.

 *DDL es como el arquitecto de la base de datos: diseña la estructura que luego se llenará de datos.*

4. DML – Lenguaje de Manipulación


- Permite **insertar, modificar y borrar datos**.

Ejemplo:

```
INSERT INTO empleados (id, nombre, salario) VALUES (1, 'Lucía',  
2500.00);
```

Las **window functions** (funciones de ventana) son poderosas para análisis:

```
SELECT nombre, salario, AVG(salario) OVER (PARTITION BY  
departamento)  
FROM empleados;
```

 *DML es el lenguaje que “da vida” a las estructuras creadas por DDL.*

5. DQL – Consultas

- **SELECT** es el corazón de SQL: **extrae información**.
- Soporta:
 - **JOINS** para combinar tablas.
 - **Subconsultas** y **CTEs** para modular las consultas.
 - Funciones de agregación (**SUM**, **AVG**, **COUNT**, etc.).
 - Filtros (**WHERE**, **HAVING**, **GROUP BY**, etc.).

 *Aprender a hacer consultas eficientes es la clave para dominar SQL.*

6. DCL – Control de Acceso


- Gestiona **quién puede hacer qué** con los datos.

Ejemplo:

```
GRANT SELECT ON clientes TO usuario_lectura;
```

- Funciona con roles y privilegios.
- Seguridad moderna:

- Filtrado por fila (**Row Level Security**).
- Ocultamiento de columnas (**Column Masking**).

 *Importante en entornos multiusuario y para proteger datos sensibles.*

7. TCL – Transacciones

- Agrupan operaciones que deben ejecutarse juntas (todo o nada).

Ejemplo:

```
BEGIN;
UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1;
UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

- Evita inconsistencias en operaciones críticas como transferencias bancarias.

 *Con TCL garantizamos la integridad y fiabilidad del sistema.*

8. SQL embebido e integración

- SQL se puede **insertar dentro de lenguajes como C, Java o Python**.
- Es útil para aplicaciones que gestionan lógica compleja y consultas a la vez.
- También se integra con **ORMs** como Django, Hibernate, etc.

 *Permite conectar la lógica del programa con la base de datos de forma fluida.*

9. Alternativas y extensiones a SQL (con ejemplos)

♦ NoSQL (Not Only SQL)

 *Diseñado para grandes volúmenes de datos, estructuras flexibles y alta disponibilidad.*

MongoDB (modelo documental)


Almacena documentos JSON, ideal para datos jerárquicos.

```
db.usuarios.find({ "direccion.ciudad": "Madrid" })
```


Cassandra (modelo columnar)

Diseñado para escritura rápida y alta disponibilidad.

```
SELECT * FROM clientes WHERE id = '123';
```

 *NoSQL es útil cuando necesitas escalabilidad horizontal y no necesitas tantas garantías de consistencia inmediata (BASE).*

♦ NewSQL

 *Combina la sintaxis SQL con capacidades de escalado horizontal (clustering, replicación, consistencia fuerte).*

CockroachDB

Usa SQL estándar y soporta ACID en entornos distribuidos.

```
SELECT COUNT(*) FROM pedidos WHERE fecha > CURRENT_DATE - INTERVAL '30 days';
```

- **TiDB** (MySQL compatible)
Similar a PostgreSQL o MySQL, pero pensado para Big Data.

🧠 *NewSQL es la opción cuando quieres escalabilidad sin renunciar a la solidez de las transacciones SQL.*

♦ GraphQL

➡ *Lenguaje de consulta flexible que permite solicitar solo los datos deseados (API modernas).*

Ejemplo de consulta:

```
{
  usuario(id: "1") {
    nombre
    pedidos {
      producto
      precio
    }
  }
}
```

Se puede convertir internamente en SQL con resolvers:

```
SELECT nombre FROM usuarios WHERE id = 1;
```

```
SELECT producto, precio FROM pedidos WHERE usuario_id = 1;
```

🧠 *GraphQL es ideal para frontends, ya que evita el overfetching o underfetching de datos.*

♦ Lenguajes específicos de dominio (DSLs)

➡ *Ofrecen funcionalidades orientadas a tareas muy concretas, como análisis estadístico o procesamiento en tiempo real.*

Pandas (Python)

Manipulación de datos similar a SQL pero en memoria.

```
df.groupby("ciudad")["ventas"].sum()
```

dplyr (R)

Operaciones similares a SQL en entornos estadísticos.

```
clientes %>% filter(ciudad == "Madrid") %>% summarise(total =  
sum(ventas))
```

Flink SQL o Kafka Streams

Para procesamiento de flujos en tiempo real con sintaxis SQL:

```
SELECT userId, COUNT(*) FROM ClickStream GROUP BY userId;
```

Estos DSLs son ideales para tareas analíticas avanzadas y procesamiento en streaming.

10. Aplicaciones modernas de SQL

- En **BI** se usa para alimentar dashboards (Power BI, Tableau).
- En **Learning Analytics** ayuda a entender cómo aprende el alumnado.
- En **IA y Big Data**, SQL sigue siendo útil para preparar datasets y combinar con Spark o TensorFlow.



Saber SQL te da acceso a muchos sectores de alta demanda: educación, salud, empresa, IA.

11. Conclusión

- SQL es robusto, universal y sigue evolucionando.
- Entender no solo su sintaxis, sino su estructura interna (subtipos, transacciones, optimización) es clave.
- Combinado con seguridad, integración y rendimiento, es una herramienta imprescindible para cualquier perfil técnico.



Idea General

El alumnado se convierte en un equipo de ingenieros de datos encargado de optimizar un sistema de base de datos existente. A partir de un conjunto de consultas SQL “defectuosas” o ineficientes, deben detectar errores, reescribir las consultas, justificar los cambios y proponer mejoras estructurales.

Tema 44. Técnicas y Procedimientos para la Seguridad de los Datos

1. Introducción

La seguridad de los datos no es solo una cuestión técnica, sino estratégica: en una era digital, perder datos equivale a perder reputación, dinero y control. El objetivo es garantizar cinco principios fundamentales:

- **Confidencialidad:** que solo personas autorizadas accedan a los datos.
- **Integridad:** que los datos no se modifiquen de forma no autorizada.
- **Disponibilidad:** que los datos estén accesibles cuando se necesiten.
- **Autenticidad:** que podamos comprobar la identidad de quien accede o modifica.
- **No repudio:** que nadie pueda negar haber realizado una acción.

Esto se aborda desde dos frentes:

- **Seguridad activa:** prevenir y detectar.
- **Seguridad pasiva:** reaccionar y recuperar.

2. Servicios de Seguridad Aplicados a los Datos

Confidencialidad

- Usamos cifrado en tránsito (TLS/SSL) y en reposo (AES, RSA).
- Clasificamos la información (pública, interna, confidencial...).
- Aplicamos modelos de acceso como **RBAC** (por roles) o **Zero Trust** (nadie es confiable por defecto).

Integridad

- Los algoritmos hash (SHA-256) detectan si los datos han cambiado.
- Las firmas digitales aseguran que un documento no ha sido alterado y que viene del remitente que dice ser.
- Sistemas como Tripwire detectan cambios no autorizados en archivos clave.
- En bases de datos usamos restricciones como **CHECK** o **UNIQUE** para asegurar reglas.

Disponibilidad

- Backup 3-2-1: tres copias, dos tipos de soporte, una externa.
- Clústeres y replicación aseguran continuidad en caso de fallo.
- Planes de recuperación garantizan que ante un incidente, no se paralice la organización.

Autenticidad y No repudio

- Autenticación fuerte con MFA (ej. app móvil o biometría).

- Timestamping con firma digital: prueba que un archivo existía en un momento determinado.

3. Técnicas de Protección de Datos

Cifrado avanzado

- En la nube usamos claves propias (BYOK) o hardware seguro (HSM).
- En bases de datos: **TDE** cifra todo el fichero; **Always Encrypted** cifra columnas sensibles.

DLP (Prevención de pérdida de datos)

- Herramientas que detectan y bloquean filtraciones por email, USB, etc.

Enmascarado y pseudonimización

- Enmascaramos datos reales en entornos de pruebas.
- La pseudonimización cumple la ley al impedir identificar directamente a una persona.

Gestión de accesos

- Aplicamos el principio del mínimo privilegio.
- Usamos plataformas como Azure AD para centralizar el control.
- Revisamos accesos regularmente y separamos funciones críticas.

4. Sistemas de Protección de Datos

Backups

- Usamos backups completos o incrementales según el caso.
- Snapshots nos permiten versiones rápidas, ideales para recuperar estados anteriores.

Monitorización y auditoría

- Herramientas como Wazuh o Splunk analizan eventos y detectan patrones anómalos.
- Se conservan logs para auditorías forenses.

Bases de datos

- Propiedades ACID garantizan seguridad en transacciones.
- Usamos triggers para auditar o validar en tiempo real.
- Consultas parametrizadas y ORMs previenen inyecciones.
- Sistemas NoSQL también incluyen roles, cifrado y validación de esquemas.

Archivos y sistemas

- Cifrado completo del disco (ej. BitLocker).

- Permisos finos (ACLs), snapshots para recuperación.

Clústeres y replicación

- Replicación síncrona = máxima consistencia, pero más lenta.
- Replicación asíncrona = mejor rendimiento, pero con posible pérdida parcial.

5. Normativa y Estándares

- **ISO 27001/27002**: referencia mundial para la gestión de seguridad.
- **NIST 800-53 y 800-207**: orientado a agencias y Zero Trust.
- **RGPD y LOPDGDD**: regulan los derechos de los ciudadanos.
- **ENS**: exige medidas específicas para sistemas públicos en España.
- Usamos metodologías como **MAGERIT** y herramientas como **PILAR** para analizar riesgos.

6. Amenazas comunes

- Ransomware, malware, empleados desleales, errores humanos...
- Fallos en la nube, servicios no autorizados (Shadow IT).
- Ataques como SQL Injection, XSS, MITM.

7. Seguridad en la Nube

- **Riesgos**: buckets públicos, claves expuestas, pérdida de control.
- **Controles**: monitoreo (CloudTrail), IAM, cifrado gestionado por el cliente, CSPM como Prisma Cloud.

8. Formación y Cultura de Seguridad

- Clasificación, políticas internas, planes de recuperación.
- Formación constante: simulacros, ejercicios.
- Programas públicos como los de INCIBE para formar a usuarios.

9. Cierre Reflexivo

La seguridad no es un estado, sino un proceso continuo. Es técnica, sí, pero también cultural y ética. Como defensores de los sistemas, nuestra misión es anticiparnos, educar y proteger.

“Guardianes del Dato: Seguridad y Protección en Entornos Reales”, integrada en el módulo de *Seguridad y Alta Disponibilidad* del CFGS de Administración de Sistemas Informáticos en Red. A través de un enfoque práctico y basado en retos, el alumnado se enfrentará al diseño y despliegue de un sistema seguro de gestión de datos, aplicando técnicas de cifrado, control de accesos, auditoría y backup en entornos simulados. Se trabajarán herramientas reales como GPG, Wazuh y sistemas de cifrado en bases de datos, y se analizarán casos emblemáticos de ciberataques para entender tanto las causas como las soluciones adoptadas. Esta propuesta no solo fomenta el dominio

técnico, sino también el pensamiento crítico, la toma de decisiones informada y el compromiso con la protección de la privacidad, todo ello alineado con la normativa vigente como el RGPD y el ENS.

Tema 54: Diseño de Interfaces Gráficas de Usuario (GUI)

I. Fundamentos del Diseño Centrado en el Usuario

1. ¿Qué es una interfaz?

Una **interfaz** es el medio a través del cual interactuamos con la tecnología. Aunque solemos pensar en pantallas, una interfaz puede ser también un botón físico, un asistente de voz o incluso una proyección en realidad aumentada. La clave está en **cómo se comunican humano y máquina**.

2. Tipos de interfaz actuales:

Aquí debes mostrar que el diseño de interfaces ha evolucionado más allá de las GUIs clásicas:

- **GUI tradicionales:** lo que ves en una app móvil, web o escritorio.
- **Conversacionales:** como hablar con Siri o chatear con un bot.
- **Gestuales/sin contacto:** mover las manos en el aire y que el sistema lo entienda.
- **AR/VR/MR:** realidad aumentada, virtual o mixta; por ejemplo, usar unas gafas para ver cómo quedará un mueble en tu casa.
- **BCI (Brain-Computer Interfaces):** investigaciones donde se controla un ordenador con la mente.

Esto demuestra tu conocimiento de interfaces emergentes y tu capacidad para anticiparte a los cambios tecnológicos.

II. Aplicación del Diseño: Del Concepto al Prototipo

3. Usabilidad y UX/UI

Es esencial dejar clara esta distinción:

- **Usabilidad:** que el sistema se use fácil y bien. Se mide con eficacia, eficiencia y satisfacción. Ejemplo: ¿puede alguien sin experiencia pedir comida por una app sin frustrarse?
- **UX (Experiencia de Usuario):** va más allá. Es cómo se **siente** esa experiencia. ¿Fue agradable? ¿Volvería a usarlo?

También es esencial mencionar la **accesibilidad**, especialmente en oposiciones: las interfaces deben ser utilizables por todos, incluidas personas con discapacidades. Esto es obligatorio legalmente en muchas plataformas públicas (WCAG 2.2).

Dark Patterns

Diseños que **engañan al usuario** para que haga algo que no quiere (como suscribirse sin darse cuenta). Importante denunciarlos y evitarlos.

4. Principios clave de diseño

- **Heurísticas de Nielsen:** son "reglas de oro" del diseño. Por ejemplo, el sistema debe mostrar siempre qué está haciendo (feedback), permitir deshacer errores, dar control al usuario, etc.
- **Leyes de diseño cognitivo:**
 - *Ley de Fitts:* cuanto más grande y más cerca esté un botón, más fácil es hacer clic. Ejemplo: el botón de cerrar debe estar en la esquina y bien visible.
 - *Ley de Hick:* más opciones generan más tiempo de decisión. Por eso simplificamos menús.
- **Gestalt:** cómo nuestra mente organiza visualmente lo que ve. Agrupamos por cercanía, color o forma. Esto se usa para crear jerarquía visual sin necesidad de escribir instrucciones.

5. Diseño visual moderno

No basta con que funcione: debe ser atractivo y claro.

- **Color:** tiene carga emocional (el rojo alerta, el verde tranquiliza). Depende también de la cultura.
- **Tipografía:** ayuda a priorizar, dirigir la lectura, generar ritmo.
- **Microinteracciones:** por ejemplo, cuando pasas el ratón por encima y el botón cambia sutilmente. Son detalles que hacen más natural y comprensible la interfaz.

6. El proceso de diseño UX/UI

Un buen diseño no nace de la inspiración, sino del método:

- **Design Thinking:** entender al usuario, definir sus problemas, idear soluciones, prototiparlas y testearlas.
- **Lean UX:** reducir tiempo y errores con pruebas rápidas e iteraciones.
- **Técnicas clave:**
 - *Personas:* imaginar usuarios tipo con nombre, edad, motivaciones.
 - *User Journey:* mapear todo lo que hace un usuario desde que entra hasta que termina su tarea.
 - *Wireframes y mockups:* bocetos digitales que muestran la estructura visual.
 - *Prototipos interactivos:* ya se puede "tocar", probar con usuarios reales y mejorar.

III. Implementación Técnica y Tendencias Futuras

7. Tipos de interfaces por plataforma

Aquí se busca mostrar que no todos los diseños se construyen igual:

Web

- *HTML5 + CSS + JS:* la base.
- *Frameworks:* React, Vue, Angular. Permiten construir interfaces reactivas y modulares.
- *PWA:* apps web que parecen nativas, sin instalar.

Móviles

- *Material Design (Android) y HIG (Apple):* guías oficiales para coherencia visual.

- *React Native / Flutter*: una sola base de código para múltiples plataformas.
- *Nativo*: más control, pero más costoso.

Escritorio

- Electron (JS para apps como Slack o VS Code), WPF (Windows), Qt (Linux/Mac).

Emergentes

- Wearables (smartwatches), Smart TVs (interfaz simple y remota).
- **AR/VR**: cada vez más relevante.
- **Interfaces por voz**: Alexa, asistentes con IA.

8. Tendencias futuras

Este apartado demuestra visión estratégica:

- **Multimodalidad**: usamos voz, tacto, gestos... a la vez.
- **Interfaces generadas por IA**: sistemas como Framer AI diseñan sin intervención humana.
- **UX personalizada por ML**: el sistema detecta tus preferencias y adapta la interfaz.
- **Diseños que se reconfiguran en tiempo real** según contexto y usuario. Una auténtica revolución.



Cierre reflexivo

El diseño de interfaces no es decoración: es una cuestión de **acceso, ética y humanidad**. Una buena GUI no se nota. Simplemente permite que las personas logren lo que necesitan sin fricciones. El mejor diseño es el que **empodera al usuario sin distraerlo**.

Título de la propuesta didáctica:

“Del Boceto al Código: Diseñando Interfaces Humanas en Entornos Reales”

Descripción de la idea:

Esta unidad didáctica se integra en el módulo de *Entornos de Desarrollo* del CFGS en Desarrollo de Aplicaciones Multiplataforma (DAM), con un enfoque interdisciplinar entre programación, diseño visual y experiencia de usuario. El alumnado abordará el reto de diseñar e implementar una interfaz gráfica funcional, accesible y usable para una aplicación real (web o móvil). Partiendo de la metodología **Design Thinking**, pasarán por todas las fases del proceso: análisis de usuarios, diseño de wireframes, prototipado interactivo con herramientas como Figma, e implementación con frameworks modernos (React, Flutter, etc.). Se incluirán también análisis de casos reales de buenos y malos diseños, con especial atención a la accesibilidad y la ética del diseño (evitar dark patterns). El proyecto culminará con una presentación grupal y pruebas de usabilidad con usuarios reales, fomentando así tanto las competencias técnicas como las comunicativas, creativas y reflexivas.

Tema 60: Sistemas basados en el conocimiento. Representación del conocimiento. Componentes y arquitectura.

1. Introducción: ¿Qué son los SBC y cómo han evolucionado?

Un **Sistema Basado en Conocimiento (SBC)** es un tipo de sistema inteligente que razona utilizando conocimiento explícito, normalmente representado en forma de reglas, hechos, ontologías o grafos. Su meta es emular la toma de decisiones humanas de forma **justificable, trazable y explicable**.

Evolución histórica:

- **Primera generación:** Sistemas expertos clásicos como *MYCIN* o *DENDRAL*. Usaban lógica simbólica: reglas del tipo “*si... entonces*” codificadas por expertos. **Limitaciones:** eran rígidos, difíciles de mantener, no aprendían de nuevos datos.

La revolución híbrida:

- **Segunda generación:** Los SBC evolucionan integrando **Machine Learning** para tratar datos complejos, mejorar la adaptabilidad y actualizar el conocimiento automáticamente.
- Esta fusión da lugar a sistemas **neuro-simbólicos**, donde los modelos de aprendizaje supervisado o no supervisado enriquecen la inferencia lógica con patrones extraídos de los datos.

2. Representación del Conocimiento: de reglas a vectores semánticos

Representar conocimiento implica **estructurar y codificar la información** de forma útil para la inferencia o el aprendizaje.

Enfoques simbólicos:

- **Ontologías** (como OWL, SNOMED CT): Estructuras jerárquicas que definen conceptos y relaciones.
- **Reglas lógicas:** Permiten deducción precisa, como en *CLIPS* o *Prolog*.
- **Grafos de conocimiento:** Google, Amazon o Facebook los usan para relacionar entidades (personas, productos, lugares).

Integración con ML:

- **Embeddings semánticos:** Algoritmos como *Word2Vec*, *BERT* o *RDF2Vec* transforman conceptos en vectores que capturan relaciones semánticas.
- Esto permite a los sistemas *razonar* con ML (similitud, clustering, clasificación) sin perder la conexión con estructuras formales.

3. Arquitectura moderna de SBC: lógica, aprendizaje y explicación

Componentes clave:

1. **Captura del conocimiento:** Puede ser manual (expertos) o automática (ML desde textos, logs o imágenes).
2. **Base de conocimiento (KB):** Contiene reglas, hechos, estructuras semánticas.
3. **Motor de inferencia:** Usa lógica formal para razonar.
4. **Módulo de Machine Learning:**
 - Detecta patrones no codificables con reglas.
 - Clasifica, predice, agrupa datos.
 - Permite razonamiento probabilístico.
5. **Módulo explicativo (XAI):** Esencial para entornos sensibles. Herramientas como *LIME*, *SHAP* o *Trepan* explican decisiones complejas.

Flujo híbrido típico:

- Extraemos reglas y hechos.
- Aplicamos lógica simbólica.
- Refinamos o complementamos con ML.
- Fusionamos resultados y explicamos al usuario.

4. Machine Learning en SBC: Sinergias clave

4.1 Extracción automática de conocimiento

- *Clustering*, *árboles de decisión* y *regresión* permiten generar reglas directamente desde datos históricos (ej: reglas para mantenimiento predictivo en fábricas).
- *LLMs fine-tuned* (como GPT-4 adaptado) pueden generar ontologías automáticamente desde bases documentales.

4.2 Nuevos motores de inferencia

- **Programación probabilística** (*Pyro*, *Stan*): Mezcla estadística y lógica para razonar bajo incertidumbre.
- **DeepProbLog**: redes neuronales que integran conocimiento lógico.
- **Neuro-symbolic AI**: como *AlphaGeometry* o el *Concept Learner*, combinan visión artificial y razonamiento simbólico.

4.3 NLP y conocimiento estructurado

- Sistemas como *spaCy* o *ChatGPT* capturan entidades clave y relaciones desde lenguaje natural.
- Se estructuran luego en grafos u ontologías que alimentan al SBC.
- Visualizaciones con *Graphviz* o *D3.js* ayudan a entender el razonamiento.

5. Aplicaciones reales de SBC + ML

Salud

- *Mayo Clinic*: sistema híbrido que combina reglas médicas, redes neuronales y NLP → mejora del 37% en precisión diagnóstica.

Banca y seguros

- Reglas regulatorias (ej. GDPR), scoring crediticio con *XGBoost*, explicaciones automáticas con *LIME* para validación legal.

Industria 4.0

- Predicción de fallos con *LSTM* + reglas de mantenimiento.
- Gemelos digitales enriquecidos con grafos semánticos.

6. Tendencias emergentes

Generative AI + SBC

- *GPT* + *Pinecone* / *Weaviate*: para búsqueda semántica sobre bases de conocimiento.
- *RAG*: mejora la veracidad de respuestas generativas consultando bases estructuradas.

Aprendizaje continuo

- Detección de *data drift* y *concept drift* → el sistema reajusta reglas con AutoML.

Quantum AI simbólica

- Proyectos en *IBM Q*, *Xanadu AI* investigan cómo simular inferencias lógicas con redes cuánticas.

7. Ejemplo práctico en Python (explicado)

```
from pyke import knowledge_engine
from sklearn.ensemble import RandomForestClassifier

# Motor de reglas simbólicas
engine = knowledge_engine.engine(__file__)
engine.activate('diagnostico')

# Entrenamos un modelo ML para datos no estructurados
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Función que combina reglas + predicción
def diagnosticar(paciente):
    reglas = engine.prove_1('diagnostico', paciente)
    pred_ml = model.predict([paciente['datos']])
```

```
return combinar(reglas, pred_ml)
```

 Este enfoque permite:

- Justificación lógica si hay reglas aplicables.
- Apoyo probabilístico si los datos son ambiguos.
- Integración con *LangChain*, *Gradio* o *Streamlit* para visualización y explicación al usuario.

8. Evaluación comparativa

Métrica	SBC Clásico	SBC + ML Integrado
Precisión diagnóstica	70%	>90%
Adaptabilidad	Baja	Alta (aprendizaje continuo)
Coste de mantenimiento	Alto	Medio (AutoML)
Explicabilidad	Muy alta	Alta (XAI)
Escalabilidad	Limitada	Alta (cloud-native)

9. Cierre reflexivo

“En sistemas donde el coste del error es alto, la transparencia no es un lujo, es una exigencia. Los SBC actuales, al combinar razonamiento lógico, aprendizaje automático y explicación estructurada, nos permiten construir tecnologías que no solo actúan con precisión, sino con responsabilidad.”

Título de la actividad:

“Sistemas que Aprenden y Explican: Integrando Machine Learning en un Sistema Basado en Conocimiento”

Descripción para el alumnado:

En esta actividad desarrollarás un pequeño sistema inteligente capaz de tomar decisiones combinando reglas lógicas con un modelo de Machine Learning entrenado por ti. Aplicaremos esta integración a un caso realista, como un diagnóstico médico simulado o la predicción de riesgo financiero, utilizando Python, Scikit-learn y herramientas de explicación como SHAP. Aprenderás cómo representar conocimiento, entrenar modelos, fusionar ambos mundos y, sobre todo, cómo hacer que el sistema justifique sus decisiones de forma clara. El objetivo es que comprendas cómo se construyen tecnologías de IA que no solo aciertan, sino que también **se explican y se adaptan**.

Tema 61: Redes y Servicios de Comunicaciones

1. Introducción: ¿Por qué son esenciales las redes hoy?

Las redes de comunicaciones son **el sistema nervioso de la sociedad digital**. Todos los servicios críticos –sanidad, transporte, comercio electrónico, administración pública, educación– dependen de infraestructuras de red robustas, seguras y eficientes.

No sólo transportan datos: permiten que la inteligencia artificial acceda a los datos, que el cloud distribuya recursos, que el IoT coordine millones de sensores... en resumen, **hacen posible la transformación digital**.



2. Arquitectura de red: cómo se construye y cómo se comporta una red

Podemos entender una red por capas, inspirándonos en modelos como **OSI o TCP/IP**, que nos ayudan a organizar sus componentes desde lo físico hasta lo lógico y lo funcional.



Capa física y de enlace (cómo conectamos dispositivos)

- Los medios físicos pueden ser **cables (UTP, coaxial), fibra óptica** o **tecnologías inalámbricas** como WiFi, LTE, 5G o satélites.
- La **topología** de red (estrella, malla, anillo...) condiciona su **resiliencia, mantenimiento y escalabilidad**.

Ejemplo real: Un hospital moderno utiliza fibra óptica para interconectar edificios, WiFi para movilidad del personal y Bluetooth para sensores biomédicos.



Capa de red y transporte (cómo se dirigen y priorizan los datos)

- Protocolos como **IP, TCP, UDP, ICMP, ARP** permiten el enrutamiento y la entrega fiable.
- La **calidad de servicio (QoS)** se mide con parámetros como **latencia, jitter o pérdida de paquetes**. Aquí entra en juego la **priorización del tráfico**: por ejemplo, una llamada VoIP no puede esperar.

Ejemplo: Una red industrial necesita garantizar que los sensores de una cadena de montaje reciban sus datos sin interrupciones. Por eso se prioriza ese tráfico con QoS frente a tareas no críticas.



Capa de aplicación (cómo usamos la red)

- Aquí encontramos los servicios que los usuarios utilizan:
 - **Correo electrónico**: SMTP, IMAP, con medidas de seguridad como DKIM o S/MIME.
 - **Navegación web**: HTTP/HTTPS, DNS, DNSSEC, CDNs como Cloudflare para acelerar el acceso.

- **Transferencia de archivos:** FTP, SFTP, y servicios cloud como Drive o Dropbox.
- **VoIP y mensajería:** protocolos SIP, RTP o apps como Signal.
- **IoT y cloud computing:** MQTT, AWS, Azure, servicios escalables y ubicuos.

3. Evolución tecnológica: de ARPANET a redes autogestionadas

Las redes han evolucionado desde el **telégrafo**, pasando por **ARPANET** y las redes móviles (de la 1G a la actual 5G), hacia un futuro dominado por **6G, redes cuánticas y ciudades inteligentes**.

- 5G ya ofrece latencias inferiores al milisegundo y capacidad para **conectar millones de dispositivos IoT** simultáneamente.
- 6G apunta a comunicaciones holográficas, más seguras y adaptadas al contexto con inteligencia artificial embebida.

Además, las redes ya no se configuran de forma manual. Con **SDN (redes definidas por software)** y **Machine Learning**, son capaces de:

- Detectar patrones de tráfico.
- Predecir fallos.
- Reconfigurarse automáticamente.

Ejemplo: Google B4 usa IA para optimizar su backbone global.

4. Seguridad en redes: del blindaje físico al modelo Zero Trust

La ciberseguridad no es un añadido, es parte del diseño. Las redes deben protegerse frente a:

- **Amenazas tradicionales:** DDoS, sniffing, spoofing, ransomware.
- **Amenazas modernas:** ataques dirigidos a dispositivos IoT, vulnerabilidades en 5G o redes industriales.

Las defensas incluyen:

- **Firewalls, IDS/IPS, cifrado (TLS, IPsec), autenticación (802.1X).**
- **Modelos Zero Trust y microsegmentación:** cada segmento se verifica de forma independiente.
- **Análisis con Machine Learning:**
 - Sistemas que detectan tráfico anómalo automáticamente.
 - Algoritmos que detectan “data drift” y ajustan reglas de seguridad.

Normativa aplicable: **GDPR, LOPDGDD, ENS**, estándares como **ISO/IEC 27001 y NIST SP 800-53** garantizan cumplimiento y buenas prácticas.

5. Gobernanza, legislación y ética de las comunicaciones

- La **Ley General de Telecomunicaciones** regula el espectro, las obligaciones de los operadores y la interoperabilidad.
- **Neutralidad de la red**: impide que se prioricen o bloqueen servicios en función de intereses comerciales.

Además, hoy hablamos de **soberanía digital**: ¿quién controla las infraestructuras? ¿qué datos pueden salir de un país? ¿cómo equilibramos innovación y privacidad?

6. Conclusión y visión de futuro

Las redes modernas deben ser:

- **Escalables** (para soportar millones de dispositivos),
- **Seguras** (para resistir ataques),
- **Resilientes** (para seguir operando incluso tras un fallo),
- **Inteligentes** (capaces de aprender y autorregularse).

Las tendencias más relevantes incluyen:

- **Redes 6G con IA integrada.**
- **Redes cuánticas** con comunicaciones imposibles de interceptar.
- **Smart cities** con redes contextuales y adaptativas.
- **Automatización basada en aprendizaje automático**, que permitirá redes que se ajusten solas a las necesidades de cada momento.

Cierre reflexivo

“En la era digital, una red no es solo un conjunto de cables y señales: es una infraestructura viva que permite que la inteligencia artificial piense, que el conocimiento se comparta y que los servicios lleguen a cada rincón del mundo. Diseñar, proteger y evolucionar esas redes no es solo un reto técnico: es una responsabilidad social.”

Título de la actividad:

“Misión: Conecta tu Mundo – Diseña y Simula tu Propia Red desde Cero”

Descripción para el alumnado:

En esta actividad te convertirás en el arquitecto de tu propia red de comunicaciones. Tendrás el reto de diseñar la red de una pequeña ciudad inteligente o un campus educativo usando herramientas como Packet Tracer o simuladores online. Deberás decidir qué dispositivos usar, cómo conectarlos, qué servicios incluir (WiFi, correo, VoIP, DNS...), y cómo protegerlos. A lo largo del proceso trabajarás en equipo, resolverás incidencias de red simuladas y competirás para ver qué grupo logra la red más eficiente, segura y creativa. Una forma práctica, visual y divertida de entender cómo funciona Internet... desde dentro.

Tema 62: Arquitecturas de sistemas de comunicaciones: capas y estándares

1. Introducción: ¿Por qué necesitamos arquitecturas?

Las comunicaciones digitales actuales –desde enviar un mensaje por WhatsApp hasta operar un dron en remoto– implican procesos complejos entre dispositivos heterogéneos. Sin un marco común, la interoperabilidad sería imposible. Aquí es donde nacen las **arquitecturas de sistemas de comunicaciones**: estructuras organizadas que permiten **que todos los dispositivos y servicios hablen el mismo idioma**.

Estos modelos definen qué funciones se deben cumplir, cómo se dividen, cómo se conectan... y todo eso sin depender de un fabricante específico. Por eso son **la base invisible pero esencial** del Internet moderno.

2. Modelo de capas: una solución modular al caos

Las arquitecturas en niveles, también llamadas **por capas**, responden a una idea simple: **dividir para entender y controlar**.

Imaginemos enviar una carta:

- La escribes (datos).
- La metes en un sobre (formato).
- Le pones dirección (direccionamiento).
- La lleva un cartero (transporte).
- Llega a su destino y se abre (aplicación).

Eso mismo hacen las capas de red: **cada una se encarga de una parte del proceso, sin conocer los detalles de las demás**.

3. Modelos de referencia: OSI vs. TCP/IP

3.1. Modelo OSI (ISO/IEC 7498)

Es un modelo teórico y didáctico de **7 capas**, cada una con funciones claras:

1. **Física**: cables, señales, conectores.
2. **Enlace de datos**: direcciones MAC, control de errores (Ethernet).
3. **Red**: rutas y direcciones lógicas (IP, ICMP).
4. **Transporte**: control de flujo y errores extremo a extremo (TCP/UDP).
5. **Sesión**: gestiona conexiones persistentes.
6. **Presentación**: formato, cifrado, compresión (TLS).
7. **Aplicación**: interacción con el usuario final (HTTP, SMTP, FTP...).

3.2. Modelo TCP/IP (Internet)

Más simple y práctico, usado en Internet. Tiene **4 capas**:

1. Acceso a red (física + enlace).
2. Internet (IP).
3. Transporte (TCP, UDP).
4. Aplicación (todos los servicios).

Comparación:

- OSI es más detallado, pero poco aplicado en productos reales.
- TCP/IP es la base funcional de Internet y sus protocolos (RFC).

4. Arquitecturas específicas por entorno

Las arquitecturas por capas se adaptan a contextos concretos:

4.1. Redes empresariales y LAN/WAN

- Ethernet, VLAN, STP, routing, NAT.
- Uso de topologías estrella o híbridas.

4.2. Redes móviles

- **4G**: eNodeB, EPC (MME, SGW, PGW).
- **5G**: Core 5G, slices (redes virtuales adaptadas a servicios), edge computing para baja latencia.

4.3. Redes industriales y IoT

- Arquitectura ISA-95: desde sensores (nivel 0) hasta gestión empresarial (nivel 4).
- Protocolos como Modbus, OPC-UA, MQTT.

4.4. Redes definidas por software (SDN)

- Se separa el control (decisiones) del plano de datos (movimiento).
- Arquitectura flexible con controladores centralizados y APIs abiertas.

5. Estandarización: hablar un mismo lenguaje digital

5.1. ¿Por qué necesitamos estándares?

- Garantizan **interoperabilidad, seguridad y compatibilidad**.
- Permiten que un dispositivo de Japón funcione con uno de Alemania sin reprogramación.

5.2. Organismos clave

- **ISO**: estándares globales (ej. modelo OSI).
- **IEEE**: redes locales y personales (Ethernet 802.3, WiFi 802.11).
- **IETF**: protocolos de Internet (IP, TCP, DNS – definidos como RFCs).

- **ITU-T**: estándares de telecomunicaciones (ej. G.711 para voz).
- **ETSI**: normas europeas, clave en 5G.

5.3. Estándares destacados

- **IEEE 802.x**: Ethernet, WiFi, Bluetooth.
- **RFCs**: documentos vivos que definen los protocolos que usamos.
- **Seguridad**: TLS, IPSec, WPA3.
- **IoT y datos ligeros**: MQTT, CoAP.

6. Ventajas del modelo por capas

- **Independencia**: cambiar la capa física (fibra por 5G) no afecta a las demás.
- **Facilita la actualización**: puedes cambiar el navegador sin alterar el cableado.
- **Mejora la seguridad**: puedes cifrar en distintas capas.
- **Interoperabilidad universal**: la red funciona sin importar el fabricante.

7. Aplicaciones reales y visión transversal

7.1. Escenario práctico: abrir una web

1. Escribes la URL (capa 7).
2. Se realiza una petición HTTP/HTTPS (capa 7).
3. Se traduce el nombre a IP vía DNS (capa 7).
4. Se establece conexión TCP (capa 4).
5. Se enruta el paquete (capa 3).
6. Se envía físicamente (capa 1).

7.2. Interacción con seguridad y legislación

- La **capa 6** gestiona el cifrado (TLS).
- La **capa 4** detecta pérdida o corrupción de datos.
- Los estándares como **TLS 1.3** o **IPsec** garantizan cumplimiento normativo (ej. ENS o GDPR).

8. Conclusión y reflexión

Las arquitecturas por niveles son **una abstracción poderosa** que permite gestionar la complejidad del mundo digital. Entenderlas es esencial para:

- Diseñar sistemas escalables y seguros.
- Resolver problemas (saber en qué capa está el fallo).
- Innovar sin romper lo ya construido.



Cierre reflexivo

“Si Internet funciona cada vez que abres una web o haces una videollamada, es porque miles de dispositivos, protocolos y sistemas, diseñados por personas que nunca se han conocido, siguen las mismas reglas. Esas reglas están

organizadas en capas, y esas capas son la base de la comunicación digital universal.”

Título de la actividad:

“El Gran Juego de las Capas: Simula Internet en el Aula”

Idea de la propuesta didáctica:

El alumnado se convertirá en las distintas capas del modelo OSI o TCP/IP y representará físicamente cómo viaja un mensaje a través de una red. Usando sobres, tarjetas, códigos y obstáculos, cada grupo actuará como una capa: la capa física entregará paquetes físicos, la de red decidirá rutas, la de transporte verificará errores, y la de aplicación leerá y muestra el mensaje. El reto incluye simular fallos, cifrados, y reenrutamientos en tiempo real. A través de esta dinámica gamificada, comprenderán de forma visual, práctica y divertida cómo funciona una arquitectura por niveles, qué responsabilidades tiene cada capa, y por qué el modelo modular es esencial para la comunicación digital moderna.

Tema 72: Seguridad en Sistemas en Red: Servicios, Protecciones y Estándares Avanzados

1. ¿Por qué es clave la seguridad en redes?

Vivimos en un mundo donde los sistemas en red están **conectados, expuestos y en constante ataque**. Cada organización, desde una pyme hasta una administración pública, depende de sus redes para trabajar. Sin seguridad:

- Un fallo puede dejarte sin acceso (indisponibilidad).
- Alguien podría robar datos sensibles (falta de confidencialidad).
- O manipular la información (pérdida de integridad).

Por eso, toda estrategia de ciberseguridad comienza por **asegurar la red**, que es el canal por donde viajan los datos.

2. Servicios de seguridad: lo que protege el acceso y el uso

Estos servicios son como “guardias” que controlan quién entra, qué hace y cómo se monitoriza todo.

👉 Autenticación y autorización

Son los mecanismos para verificar quién eres y qué puedes hacer. Aquí usamos:

- Contraseñas, MFA (autenticación en dos pasos), biometría.
- Protocolos como Kerberos o OAuth2, y sistemas como SSO.

👉 Control de acceso

No todo el mundo debe ver todo. Por eso se aplican modelos como:

- RBAC (según rol) o ABAC (según atributos dinámicos).
- Se gestiona a través de VLANs, NACs y filtrado de tráfico.

👉 Cifrado y no repudio

Se usa para **proteger los datos durante el tránsito** (HTTPS, VPNs) o almacenados (cifrado de discos). Y para demostrar que algo fue enviado por alguien y no ha sido manipulado (firmas digitales).

👉 Auditoría y SIEM

Los sistemas de registro y correlación (como Wazuh o Splunk) permiten **ver qué ocurre en la red** y detectar anomalías.

3. Técnicas de protección: cómo se reduce el riesgo

Aquí se trata de **limitar la superficie de ataque** y endurecer el entorno.

Segmentación

Divide la red en zonas controladas (como separar contabilidad de invitados), lo que evita que un ataque se propague. Hoy se usan VLANs, SDN y microsegmentación.

Bastionado (Hardening)

Eliminar servicios innecesarios, configurar correctamente los que quedan y automatizarlo con herramientas como Ansible.

Prevención de amenazas

Implementar EDRs que detecten actividad maliciosa, usar DNS seguro (DNSSEC) y bloquear IPs sospechosas.

4. Sistemas especializados: defensa en profundidad

Los ataques pueden venir por muchos frentes, por eso se protegen varias capas:

Firewalls

Controlan el tráfico entre redes. Los modernos (Next-Gen) pueden ver qué aplicaciones lo generan y bloquear ataques en tiempo real.

IDS/IPS

Detectan (IDS) y bloquean (IPS) comportamientos maliciosos, como escaneos de puertos o intentos de intrusión.

Backups y recuperación

Imprescindible ante ransomware. Se aplica la regla 3-2-1: tres copias, dos formatos, una externa.

5. Normativas y estándares: la ciberseguridad como obligación legal

Las organizaciones no pueden improvisar. Deben cumplir con estándares reconocidos y normativas legales.

Estándares técnicos

- ISO 27001: gestión de la seguridad.
- NIST SP 800-53: catálogo de controles.
- MITRE ATT&CK: guía de técnicas ofensivas.
- COBIT: gobierno TI.

Legislación

- RGPD/LOPDGDD: protección de datos personales.
- ENS (en España): obligatorio en sector público.

- NIS2: obliga a proteger redes críticas en la UE.

Evaluación de riesgos

Usamos metodologías como MAGERIT y herramientas como PILAR para identificar y tratar amenazas.

6. Amenazas reales: ¿a qué nos enfrentamos hoy?

- **MITM**: interceptan tu tráfico si no estás cifrado.
- **Ransomware**: secuestran tus datos.
- **Ataques en la nube**: por errores de configuración.
- **DDoS**: saturan tus servicios y los tiran abajo.

Aquí no basta con prevención, se necesita **detectar, responder y recuperarse rápido**.

7. Formación y concienciación: el usuario es la primera línea

Muchos ataques empiezan con un clic mal hecho. Por eso:

- Se hacen simulaciones (phishing, ransomware).
- Se participa en iniciativas como CyberCamp o CTFs.
- Se mide el impacto con pruebas antes y después.

“Puedes tener la mejor tecnología del mundo, pero si tus usuarios no entienden los riesgos, estás en peligro.”

8. Conclusión: una visión estratégica

La seguridad en red no es solo técnica, es una **decisión organizativa crítica**. Es lo que garantiza:

- Que el negocio siga funcionando tras un incidente.
- Que los datos de las personas estén protegidos.
- Que las instituciones conserven su reputación y confianza.

Título de la actividad:

“Ciberdefensores en Red: Simula tu SOC y Defiende tu Infraestructura”

Idea de la propuesta didáctica:

El alumnado se dividirá en equipos que simulan el funcionamiento de un Centro de Operaciones de Seguridad (SOC). Cada grupo tendrá que proteger una infraestructura virtual (simulada en una herramienta como TryHackMe, VirtualBox o una red Packet Tracer) de una serie de ataques ficticios (phishing, escaneo de puertos, ransomware, DDoS). Utilizarán roles reales (analista de SIEM, responsable de red, especialista en hardening, responsable de backups) y deberán aplicar medidas como segmentación, uso de firewalls, análisis de logs y respuestas rápidas ante incidentes. Ganará el equipo que mantenga su red más estable, segura y funcional. Se combina gamificación, trabajo colaborativo y simulación práctica de conceptos reales de seguridad en red.

Tema 74: Sistemas Multimedia

Introducción

En la sociedad digital actual, los sistemas multimedia no solo permiten reproducir contenidos, sino que conforman entornos complejos donde convergen texto, imagen, sonido, vídeo y animación en tiempo real. Estos sistemas integran tecnologías de codificación, transmisión, análisis e inteligencia artificial, y son esenciales en ámbitos tan diversos como la educación, el entretenimiento, la sanidad, la industria o las ciudades inteligentes. Comprender su arquitectura, funcionamiento y evolución permite diseñar experiencias digitales interactivas, eficientes y éticamente responsables, adaptadas a un entorno tecnológico en continua transformación.

Representación Digital de Medios

▪ Imagen

- **Raster vs Vectorial:** Las imágenes **raster** (JPEG, PNG, BMP) están formadas por píxeles; se distorsionan al ampliarse. Las **vectoriales** (SVG) están definidas por ecuaciones geométricas, por lo que escalan perfectamente.
- **Canal alfa:** Permite representar transparencia, muy útil en superposiciones de imágenes.

▪ Audio

- **Tasa de muestreo:** Cuántas veces por segundo se toma una muestra del sonido analógico para digitalizarlo (ej. 44.1 kHz en audio CD).
- **Bitrate:** Cantidad de datos por segundo; más alto implica mayor calidad pero mayor tamaño.
- **Formatos:**
 - **WAV:** sin compresión.
 - **FLAC:** comprimido sin pérdida.
 - **AAC/MP3:** comprimidos con pérdida, optimizan tamaño sacrificando algo de calidad.

▪ Vídeo

- **FPS (Frames per Second):** Define cuántas imágenes se muestran por segundo; más FPS = mayor fluidez.
- **Códec vs Contenedor:** El **códec** (ej. H.264) comprime el vídeo, mientras que el **contenedor** (ej. MP4) agrupa audio, vídeo, subtítulos en un solo archivo.

Procesamiento y Codificación

▪ Transformadas

- **Fourier:** Descompone una señal en sus componentes de frecuencia. Muy útil en audio.

- **DCT**: Usada en JPEG y MPEG para comprimir imágenes y vídeo, aprovechando que los cambios lentos de intensidad son más relevantes visualmente.
- **Wavelets**: Permiten representar detalles a distintas escalas; se usan en JPEG2000.

▪ **Convoluciones**

Operaciones aplicadas a imágenes (mediante “filtros”) para detectar bordes, suavizar, realzar detalles. Son la base de las redes neuronales convolucionales (CNNs) usadas en visión artificial.

Streaming Adaptativo y Protocolos

▪ **HLS y DASH**

Permiten que un vídeo cambie de calidad en tiempo real según el ancho de banda del usuario. El servidor divide el vídeo en fragmentos de diferente resolución y el reproductor elige el más adecuado en cada momento.

▪ **RTMP y RTSP**

Protocolos clásicos usados en videoconferencias y cámaras IP, priorizan baja latencia sobre calidad constante.

Inteligencia Artificial Multimedia

▪ **IA Generativa**

- **Stable Diffusion / DALL·E**: modelos que generan imágenes a partir de descripciones textuales.
- **NeRF**: recrea escenas 3D fotorrealistas desde múltiples imágenes.
- **Voice Cloning**: clona voces con pocas muestras, útil pero éticamente complejo.

▪ **Análisis**

- **YOLOv8 / DETR**: detectan objetos en vídeo en tiempo real.
- **CLIP / GPT-4V**: comprenden imágenes y texto conjuntamente. Ej.: relacionar “una persona montando en bici” con una imagen correspondiente.

Herramientas y Frameworks

- **FFmpeg**: herramienta por línea de comandos para editar, convertir y procesar audio/vídeo.
- **OpenCV**: biblioteca para tareas de visión artificial, como detección facial o seguimiento de movimiento.
- **MediaPipe**: biblioteca optimizada para móviles que permite, por ejemplo, detectar gestos con las manos o seguir una cara.

Tendencias Técnicas Emergentes

- **Codificación neural (NeRF, Gaussian Splatting):** en lugar de usar píxeles, las escenas se generan a partir de redes neuronales que modelan la luz y la geometría.
- **Códecs volumétricos (V-PCC, MIV):** permiten transmitir vídeo 3D en XR o metaverso.
- **Streaming en edge y 5G:** procesamiento multimedia directamente en el dispositivo o muy cerca del usuario para reducir latencia.

Aspectos Éticos y Legislativos

- **Deepfakes:** vídeos manipulados que imitan rostros y voces. Se combate con IA y trazabilidad.
- **Sesgos:** sistemas de reconocimiento facial pueden fallar más con ciertas etnias o géneros si los datos de entrenamiento son poco diversos.
- **IA Act (UE):** regula el uso de IA, clasificándola por nivel de riesgo y exigiendo transparencia.

Título de la actividad:

 **“Youtubers por un Día: Crea, Codifica y Publica tu Propio Canal Multimedia”**

Propuesta didáctica orientada a Bachillerato (Tecnología o TIC):

El alumnado, organizado en grupos, asumirá el rol de creadores de contenido y técnicos de su propio canal multimedia educativo. Cada grupo elegirá un tema (ciencia, historia, cultura digital...), grabará un vídeo corto (2-3 min) y deberá optimizarlo para su publicación en distintas plataformas (YouTube, web, app móvil), explorando conceptos como compresión de vídeo, edición, subtítulo y formatos (MP4, WebM, códecs). Además, simularán condiciones reales de red aplicando bitrate adaptativo y analizando el resultado visual. Usarán herramientas accesibles como OBS Studio, HandBrake y CapCut, y presentarán su "canal multimedia" a la clase. Ganará el grupo con el mejor equilibrio entre creatividad, calidad visual, eficiencia técnica y claridad del mensaje. Todo en un entorno lúdico, cooperativo y altamente motivador.

Guía práctica de estrategias docentes para Informática (Apoyo)

1. Aprendizaje Basado en Proyectos (ABP)

Descripción:

El alumno trabaja en proyectos reales o simulados, aplicando los conocimientos adquiridos de manera práctica.

Ejemplo:

- **Proyecto:** Los estudiantes diseñan y desarrollan una aplicación para la gestión de bibliotecas en línea, siguiendo todas las etapas del desarrollo de software: planificación, desarrollo, pruebas y entrega final.
- **Beneficio:** El alumnado se enfrenta a problemas reales y desarrolla habilidades técnicas y de trabajo en equipo.

2. Gamificación

Descripción:

El aula se convierte en un juego, donde los alumnos reciben recompensas (puntos, insignias) por completar tareas y desafíos.

Ejemplo:

- **Juego:** Crear una serie de misiones semanales donde los estudiantes resuelven problemas de programación. Cada misión completada suma puntos, y al final del trimestre, los estudiantes con más puntos reciben premios como “Coder del mes” o “Líder en GitHub”.
- **Beneficio:** Aumenta la motivación y hace el aprendizaje más interactivo y divertido.

3. Aula Invertida (Flipped Classroom)

Descripción:

Los estudiantes aprenden nuevos contenidos de manera autónoma fuera del aula y usan el tiempo en clase para resolver dudas y trabajar en actividades prácticas.

Ejemplo:

- **Actividad:** Los alumnos visualizan un video tutorial sobre redes y seguridad informática en casa. Luego, en clase, realizan un ejercicio práctico de configuración de redes y depuración de fallos.
- **Beneficio:** Promueve la autonomía del alumno y aprovecha el tiempo de clase para consolidar lo aprendido.

4. Aprendizaje Cooperativo

Descripción:

Los estudiantes trabajan en grupos para resolver problemas o desarrollar proyectos, fomentando el aprendizaje colaborativo.

Ejemplo:

- **Actividad:** Los alumnos se agrupan para crear una página web. Cada miembro del grupo es responsable de una parte (diseño, contenido, programación), y deben colaborar para integrarlo todo en una única plataforma.
- **Beneficio:** Mejora las habilidades de comunicación, trabajo en equipo y resolución de conflictos.

5. Estudio de Casos

Descripción:

El estudiante analiza y resuelve un caso práctico basado en situaciones reales del sector profesional.

Ejemplo:

- **Caso:** Los estudiantes tienen que analizar el caso de una empresa que ha sufrido un ataque cibernético y proponer medidas de seguridad y recuperación.
- **Beneficio:** Fomenta el pensamiento crítico y la capacidad para aplicar conocimientos en situaciones del mundo real.

6. Aprendizaje Basado en Problemas (ABP)

Descripción:

Los estudiantes trabajan sobre un problema complejo, investigando y resolviendo el desafío de manera autónoma o en grupo.

Ejemplo:

- **Problema:** Los estudiantes deben solucionar una caída en un servidor de base de datos de una empresa. Deberán investigar las causas y presentar una solución.
- **Beneficio:** Fomenta la resolución de problemas y el pensamiento lógico.

7. Evaluación Formativa

Descripción:

Evaluar de manera continua el progreso de los estudiantes para ajustar la enseñanza en tiempo real.

Ejemplo:

- **Evaluación:** Durante el curso, los estudiantes entregan pequeñas tareas y pruebas de codificación que se corrigen y retroalimentan de manera constante.
- **Beneficio:** Permite identificar debilidades a tiempo y corregirlas antes del examen final.

8. Proyectos Interdisciplinarios

Descripción:

Se realizan proyectos que integran contenidos de diferentes áreas o asignaturas, fomentando la conexión de conocimientos.

Ejemplo:

- **Proyecto:** Los estudiantes de programación y los de administración de sistemas colaboran para crear una red segura y desarrollar una aplicación de gestión interna.
- **Beneficio:** Ayuda a los estudiantes a ver cómo sus conocimientos pueden aplicarse en diferentes contextos profesionales.

9. Uso de Tecnologías Educativas**Descripción:**

Incorporar herramientas digitales para mejorar la enseñanza y el aprendizaje.

Ejemplo:

- **Herramientas:** Usar plataformas como GitHub para el control de versiones y colaboración, Trello para la gestión de proyectos y Slack para la comunicación grupal.
- **Beneficio:** Mejora la organización, la colaboración y el uso de herramientas reales del sector.

10. Aprendizaje Basado en Simulaciones**Descripción:**

Los estudiantes trabajan con simuladores para practicar habilidades sin los riesgos asociados al trabajo en entornos reales.

Ejemplo:

- **Simulador:** Usar un simulador de redes para que los estudiantes practiquen la configuración de un router o la gestión de tráfico de red.
- **Beneficio:** Los estudiantes experimentan en un entorno controlado sin consecuencias reales, lo que les permite cometer errores y aprender de ellos.

11. Tutorías Personalizadas**Descripción:**

Atender de forma individual a los estudiantes, identificando sus necesidades y ayudándoles a mejorar sus habilidades de forma personalizada.

Ejemplo:

- **Actividad:** Sesiones individuales con los estudiantes para revisar proyectos de programación y guiarlos en la mejora de su código.
- **Beneficio:** Ofrece atención específica a cada alumno, mejorando su rendimiento en áreas donde necesite más apoyo.

12. Aprendizaje Basado en la Investigación**Descripción:**

Fomentar que los estudiantes investiguen de manera autónoma temas y problemas que les interesen.

Ejemplo:

- **Investigación:** Los estudiantes investigan sobre nuevas tecnologías como blockchain o IoT y desarrollan un informe detallado sobre sus aplicaciones en el mundo real.
- **Beneficio:** Desarrolla habilidades de investigación y pensamiento autónomo, preparando a los estudiantes para el aprendizaje continuo.

13. Desafíos y Hackatones

Descripción:

Organizar competencias donde los estudiantes resuelven retos técnicos en un tiempo limitado, incentivando la creatividad y el trabajo bajo presión.

Ejemplo:

- **Desafío:** Organizar una hackathon donde los estudiantes deben crear una aplicación móvil en 48 horas para resolver un problema específico de la comunidad.
- **Beneficio:** Fomenta la creatividad, el trabajo en equipo y la capacidad de resolver problemas en condiciones de presión.

14. Trabajo Colaborativo con Profesores

Descripción:

El alumnado trabaja directamente en proyectos junto a los profesores, como iguales, en un ambiente de colaboración continua. Esto les permite aprender de la experiencia y la metodología profesional de los docentes.

Ejemplo:

- **Actividad:** Los estudiantes participan en el desarrollo de un proyecto real junto a sus profesores, como la creación de una herramienta de gestión para el aula o un sistema de administración de recursos educativos. Los docentes actúan como mentores y co-creadores, guiando al alumnado en cada etapa del proceso.
- **Beneficio:** Los estudiantes desarrollan habilidades de colaboración en el entorno profesional y aprenden el proceso de trabajo real al lado de expertos, lo que les da una perspectiva más profunda del mundo laboral.

15. Aprendizaje Experiencial

Descripción:

Los estudiantes aprenden a través de la experiencia directa y la reflexión sobre ella, aplicando los conocimientos en situaciones reales o simuladas que reflejan la realidad del entorno profesional.

Ejemplo:

- **Actividad:** Los estudiantes configuran y gestionan un entorno de red para una pequeña empresa simulada, enfrentándose a desafíos como fallos en el sistema, actualización de software, o configuración de seguridad.
- **Beneficio:** Fomenta el aprendizaje a través de la acción, permitiendo que los estudiantes no solo aprendan teoría, sino también cómo aplicarla en situaciones reales o simuladas.

16. Uso de Realidad Aumentada y Virtual (AR/VR)

Descripción:

Incorporar tecnologías de realidad aumentada y virtual para crear entornos inmersivos donde los estudiantes puedan practicar habilidades de forma más interactiva y realista.

Ejemplo:

- **Actividad:** Los estudiantes usan un entorno virtual para practicar la configuración de hardware o la administración de redes. Pueden simular el montaje de ordenadores o experimentar con redes en un entorno virtual seguro.
- **Beneficio:** Proporciona una experiencia de aprendizaje mucho más dinámica y visual, permitiendo que los estudiantes practiquen sin los riesgos de un entorno real.

17. Mentorización entre Iguales

Descripción:

Fomentar que los estudiantes más avanzados ayuden a sus compañeros menos experimentados, creando una cultura de aprendizaje colaborativo y apoyo mutuo.

Ejemplo:

- **Actividad:** Establecer un sistema de mentoría en el que los estudiantes de niveles superiores asesoren y guíen a los de niveles inferiores en proyectos de programación o resolución de problemas.
- **Beneficio:** Ayuda a reforzar los conocimientos de los estudiantes más avanzados, mientras que proporciona apoyo a los estudiantes que necesitan mejorar.

18. Evaluación Auténtica

Descripción:

Evaluar el aprendizaje de los estudiantes no solo mediante exámenes teóricos, sino también a través de tareas y proyectos que reflejan el trabajo que realizarían en un entorno profesional.

Ejemplo:

- **Actividad:** Los estudiantes deben entregar un proyecto completo de software (desde el análisis de requisitos hasta el desarrollo y las pruebas), y la evaluación se basa en la calidad del producto final, la documentación y su presentación.
- **Beneficio:** La evaluación auténtica mide las competencias reales de los estudiantes, preparándolos para el mundo laboral con una evaluación más alineada con las exigencias profesionales.

19. Aprendizaje Personalizado

Descripción:

Adaptar los contenidos y las actividades a las necesidades y ritmo de aprendizaje de cada estudiante, permitiendo que cada uno avance según su capacidad y estilo de aprendizaje.

Ejemplo:

- **Actividad:** Usar plataformas de aprendizaje adaptativo que ofrezcan diferentes niveles de dificultad en los contenidos según el progreso de cada estudiante.

- **Beneficio:** Ayuda a cada estudiante a alcanzar su máximo potencial, respetando sus tiempos de aprendizaje y proporcionando desafíos adecuados a su nivel.

20. Tareas Reales y Desafíos de la Industria

Descripción:

Involucrar a los estudiantes en tareas y desafíos que provienen directamente de la industria, permitiéndoles aplicar lo aprendido en escenarios profesionales reales.

Ejemplo:

- **Actividad:** Los estudiantes trabajan con empresas locales para desarrollar soluciones a problemas específicos que estas enfrentan, como crear una página web para un pequeño comercio o desarrollar un sistema de inventarios.
- **Beneficio:** Aumenta la conexión entre lo que aprenden y cómo se aplica en el mundo real, dando a los estudiantes una experiencia muy valiosa para sus futuros profesionales.

21. Fomentar la Creatividad y la Innovación

Descripción:

Estimular la creatividad de los estudiantes, animándolos a desarrollar ideas nuevas y a experimentar con soluciones innovadoras para los problemas.

Ejemplo:

- **Actividad:** Los estudiantes deben crear una solución tecnológica creativa para un problema social o medioambiental, como desarrollar una app que ayude a reducir el desperdicio de alimentos.
- **Beneficio:** Fomenta el pensamiento innovador, la resolución creativa de problemas y el interés por mejorar la sociedad a través de la tecnología.