

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	7
Tema 3: Componentes, estructura y funcionamiento de la Unidad Central de Proceso (CPU)	12
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	25
Tema 12: Organización Lógica de los Datos – Estructuras Dinámicas	29
Tema 20: Explotación y administración de sistemas operativos monousuario y multiusuario	33
Tema 21: Sistemas informáticos. Estructura física y funcional	38
Tema 22: Planificación y explotación de sistemas informáticos. Configuración. Condiciones de instalación. Medidas de seguridad. Procedimientos de uso.	40
Tema 23: Diseño de algoritmos. Técnicas descriptivas.	42
Tema 24: Lenguajes de programación: Tipos y características	46
Tema 25: Programación Estructurada. Estructuras Básicas. Funciones y Procedimientos.	52
Tema 26: Programación modular. Diseño de funciones. Recursividad. Librerías.	55
Tema 27: Programación orientada a objetos. Objetos. Clases. Herencia. Polimorfismo. Lenguajes.	59
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.	64
Tema 32: Lenguaje C. Manipulación de estructuras de datos dinámicas y estáticas. Entrada y salida de datos. Gestión de punteros. Punteros a funciones.	68
Tema 36: La manipulación de datos. Operaciones. Lenguajes. Optimización de consultas.	72
Tema 39: Lenguajes para la definición y manipulación de datos en sistemas de Bases de Datos Relacionales. Tipos. Características. Lenguaje SQL	76
Tema 44. Técnicas y Procedimientos para la Seguridad de los Datos	79
Tema 54: Diseño de Interfaces Gráficas de Usuario (GUI)	83
Tema 60: Sistemas basados en el conocimiento. Representación del conocimiento. Componentes y arquitectura.	85
Tema 61: Redes y Servicios de Comunicaciones	90
Tema 72: Seguridad en Sistemas en Red: Servicios, Protecciones y Estándares Avanzados	93
Tema 74: Sistemas Multimedia	96
Guía práctica de estrategias docentes para Informática (Apoyo)	100

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 de la información.
- Por qué es importante: eficiencia, seguridad y compatibilidad.
- Aplicación en programación, redes y hardware.

2. Sistemas de numeración

- Conceptos básicos: base, dígitos, sistema posicional.
- Sistemas utilizados:
 - Decimal (base 10)
 - Binario (base 2)
 - Octal (base 8) y hexadecimal (base 16): conexión con binario.
- Relación entre bases y su utilidad en informática.

3. Sistema binario: representación de la información

- Bits y bytes como unidad mínima.
- Representación de:
 - Números (enteros con/sin signo CA1/CA2, punto flotante - IEEE 754).
 - Caracteres (ASCII y Unicode).
 - Otros tipos de datos (imágenes, audio, vídeo).

4. Conversiones entre bases numéricas

- Métodos de conversión: decimal \leftrightarrow binario, binario \leftrightarrow hexadecimal, etc.
- Ejercicios prácticos y utilidad real en programación, hardware y redes.

5. Operaciones y lógica binaria

- Aritmética binaria: suma, resta, multiplicación, división.
- Representación de números negativos:
 - Complemento a 1 y a 2.
- Operaciones lógicas fundamentales: AND, OR, XOR, NOT.

6. Sistemas octal y hexadecimal

- Por qué se usan en informática.
- Conversión rápida desde binario.
- Aplicaciones prácticas:
 - Direcciones de memoria.

- Debugging.
- Ensamblador.
- Visualización de datos (hex editors, hashes).

7. Códigos binarios

7.1 Códigos numéricos

- BCD (Binary-Coded Decimal).
- Código Exceso-3.
- Código Gray: reducción de errores en transiciones digitales.

7.2 Códigos alfanuméricos

- ASCII (7 y 8 bits).
- Unicode: UTF-8, UTF-16, UTF-32. Soporte internacional.

8. Detección y corrección de errores

- Necesidad de asegurar la integridad de los datos transmitidos.
- Códigos de detección:
 - Bit de paridad.
 - CRC (Cyclic Redundancy Check).
- Códigos de corrección:
 - Hamming.
 - Reed-Solomon: aplicaciones en discos, QR, transmisión digital.

9. Seguridad informática y protección de la información

9.1 Funciones hash

- Definición y características: unidireccionalidad, resistencia a colisiones.
- Algoritmos:
 - Obsoletos: MD5, SHA-1.
 - Recomendado: SHA-256.
- Aplicaciones: integridad, autenticación, contraseñas (con salts), blockchain.

9.2 Cifrado y ataques

- Amenazas:
 - Tablas Rainbow.
 - Fuerza bruta.
 - Colisiones.
- Defensa:
 - Algoritmos seguros: bcrypt, PBKDF2, Argon2.
- Criptografía:
 - Simétrica (AES).

- Asimétrica (RSA, ECC).
- Aplicaciones: HTTPS, almacenamiento, VPN.

10. Comunicación digital

10.1 Elementos de un sistema de comunicación

- Modelo de Shannon y Weaver.
- Tipos de señales: analógicas vs. digitales.
- Medios de transmisión: cableados (cobre, fibra) e inalámbricos (Wi-Fi, Bluetooth).
- Protocolos estándar: TCP/IP, Ethernet, Wi-Fi.

10.2 Compresión de datos

- **Sin pérdida:**
 - Algoritmos: Huffman, LZ77, LZ78, LZW.
- **Con pérdida:**
 - JPEG, MP3, H.264.
- Objetivo: optimizar almacenamiento y transmisión.

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.
- **Superscalaridad:** 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 empujado, 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:** bloques fijos.
 - **Segmentación:** bloques lógicos.
 - **Segmentación paginada:** combinación flexible de ambas.
- **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 (deque)

- 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 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

- Sistemas informáticos: combinación de hardware, software y redes.
- Evolución:
 - Mainframes → PCs → Cloud → Edge Computing.
- Tecnologías actuales: IA, virtualización, contenedores.

2. Sistemas Informáticos

2.1. Definición y Evolución

- Hardware + Software + Redes → Procesamiento y almacenamiento.
- Etapas:
 - 60-70: Mainframes.
 - 80-90: PCs y servidores.
 - 2000s: Cloud y virtualización.
 - 2010s: Edge, contenedores, IA.

2.2. Clasificación

- Por tamaño:
 - Microcomputadoras, servidores, supercomputadoras.
- Por arquitectura:
 - Cliente-servidor, cloud, distribuidos.
- Por finalidad:
 - Empresarial, IoT, embebidos.

2.3. Tendencias Actuales

- Cloud y Serverless (AWS Lambda).
- Contenedores (Docker, Kubernetes).
- Edge Computing (procesamiento cercano).
- Hardware especializado (GPUs, TPUs).

3. Estructura Física y Funcional

3.1. Hardware

3.1.1. Arquitectura Von Neumann:

- CPU + Memoria + E/S.
- Arquitecturas: x86, ARM, RISC-V.

3.1.2. CPU:

- Componentes: ALU, registros, núcleos (multicore).
 - Coprocesadores: GPUs, TPUs.

3.1.3. Memoria y Almacenamiento:

- RAM: DDR5, HBM.
- Almacenamiento: SSD/NVMe → Cloud (S3, Ceph).

3.1.4. Periféricos:

- Tradicionales (teclado, ratón).
- Avanzados (sensores IoT, VR).

3.1.5. Redes:

- SDN, Wi-Fi 6, Edge Computing.

3.2. Software

3.2.1. Programación:

- Lenguajes: Python, Rust, Go.
- Herramientas: VS Code, Git.

3.2.2. Aplicación:

- Empresarial: ERP, CRM.
- Cloud: SaaS (Google Workspace).

3.2.3. Sistema:

- SO: Windows, Linux, macOS.
- Cloud-native: Chrome OS, Firecracker.

3.2.4. Virtualización:

- Máquinas virtuales: VMware, KVM.
- Contenedores: Docker → Kubernetes.

3.2.5. Seguridad:

- IAM, Zero Trust, TLS 1.3.
- Monitorización: Grafana, ELK Stack.

4. Conclusiones

- Futuro:
 - Mayor uso de Edge Computing y contenedores.
 - Hardware optimizado para IA y eficiencia energética.
 - Seguridad crítica en entornos distribuidos.

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

1. Introducción

- Enfoque en la planificación, instalación, seguridad y uso de sistemas informáticos.
- Importancia de la escalabilidad, disponibilidad y cumplimiento normativo.

2. Sistemas Informáticos vs. Sistemas de Información

- 2.1. Diferencias:
 - Sistema informático: Hardware + Software.
 - Sistema de información: Datos + Procesos + Usuarios.
- 2.2. Evolución:
 - De mainframes a cloud y microservicios.
- 2.3. Arquitecturas:
 - Monolítica, cliente-servidor, microservicios.
- 2.4. Sistemas distribuidos:
 - Cloud computing (IaaS, PaaS, SaaS), Edge Computing.

3. Planificación de Sistemas

- 3.1. Diseño de infraestructura TI:
 - Requerimientos técnicos y empresariales.
- 3.2. Selección de HW/SW:
 - Balance entre coste, rendimiento y escalabilidad.
- 3.3. Estrategias de despliegue:
 - On-premise, Cloud (pública/privada), Híbrida.
- 3.4. Virtualización y contenedores:
 - VMware/KVM (VMs) → Docker/Kubernetes (contenedores).
- 3.5. Rendimiento y escalabilidad:
 - Load balancing, autoescalado (AWS Auto Scaling).

4. Explotación de Sistemas

- 4.1. Organización del departamento:
 - Roles: Administradores, DevOps, SOC.
- 4.2. Automatización (DevOps):
 - CI/CD (Jenkins, GitLab CI).
- 4.3. Monitorización:
 - Prometheus + Grafana, Zabbix, ELK Stack.
- 4.4. Alta disponibilidad:
 - Clústeres, replicación, RAID.

5. Instalación y Configuración

- 5.1. Ubicación física:

- Data Centers (tiering) vs. Edge Computing.
- 5.2. Periféricos e IoT:
 - Sensores, gateways, protocolos (MQTT).
- 5.3. Redes:
 - Cableado (CAT6/7), Wi-Fi 6, SD-WAN.
- 5.4. Seguridad física:
 - SAI (UPS), redundancia eléctrica, control ambiental (temperatura/humedad).

6. Gestión de Configuración

- 6.1. Conceptos clave:
 - CMDB (Base de Datos de Gestión de Configuración).
- 6.2. Inventario TI:
 - Herramientas: Snipe-IT, Lansweeper.
- 6.3. Automatización:
 - Ansible, Puppet, Chef.
- 6.4. Gestión de incidencias:
 - ITIL, ticketing (Jira, ServiceNow).
- 6.5. Control de versiones:
 - GitOps (Git + Kubernetes).
- 6.6. Cumplimiento:
 - ISO 27001, GDPR, ENS.

7. Seguridad en Sistemas

- 7.1. Zero Trust:
 - Verificación continua, mínimos privilegios.
- 7.2. Seguridad en redes:
 - Firewalls (Next-Gen), IDS/IPS, VPN (WireGuard).
- 7.3. Seguridad en SO/aplicaciones:
 - Parches, hardening (CIS Benchmarks).
- 7.4. Seguridad en Cloud/Contenedores:
 - Kubernetes RBAC, Docker Bench Security.
- 7.5. Amenazas avanzadas:
 - Ransomware, APTs, XSS/SQLi.

8. Procedimientos de Uso

- 8.1. Buenas prácticas:
 - Políticas de contraseñas, backups.
- 8.2. Documentación y formación:
 - Wikis (Confluence), simulacros de ciberseguridad.
- 8.3. Control de acceso:
 - IAM, MFA, LDAP/Active Directory.
- 8.4. Auditorías:
 - Logs centralizados (SIEM: Splunk, Wazuh).

9. Conclusiones

- Tendencias clave: Automatización (DevOps/GitOps), seguridad Zero Trust, hybrid cloud.
- Retos futuros: Ciberseguridad, eficiencia energética en Data Centers.

Tema 23: Diseño de algoritmos. Técnicas descriptivas.

1. Introducción

1.1 Concepto de algoritmo

Un algoritmo es una secuencia finita de instrucciones no ambiguas que permiten resolver un problema. Características clave:

- Precisión: Cada paso debe estar perfectamente definido.
- Determinismo: A igual entrada, igual salida.
- Efectividad: Todas las operaciones deben ser computables.

1.2 Importancia de los algoritmos en computación

- Son la base de la programación y la eficiencia computacional.
- Aplicaciones clave:
 - Inteligencia Artificial: Algoritmos de aprendizaje automático.
 - Ciberseguridad: Algoritmos criptográficos.
 - Big Data: Procesamiento masivo y distribuido de datos.

1.3 Evolución histórica

- Desde algoritmos clásicos como el de Euclides hasta los modernos de computación cuántica.
- Influencia del hardware:
 - GPUs y TPUs: Procesamiento paralelo.
 - Computación en la nube.
 - Algoritmos cuánticos.

2. Elementos básicos de los algoritmos

2.1 Acciones fundamentales

- Asignaciones:
Ejemplo: $x = 5 + 3$
Buenas prácticas: evitar redundancias y reutilizar variables.
- Entradas/Salidas:
Validar datos de entrada, mostrar salidas claras.

2.2 Estructuras de control

- Secuenciales:
Ejemplo: Cálculo del área de un círculo (leer radio → calcular → mostrar resultado).
- Condicionales (If-Else, Switch):
 - Usar If-Else para condiciones simples.
 - Usar Switch cuando haya múltiples casos.
 - Ordenar condiciones por frecuencia de uso.
- Iterativas (bucles):
 - For: útil cuando se conoce el número de iteraciones.
 - While: útil para condiciones dinámicas.
 - Optimización: evitar cálculos innecesarios dentro del bucle.

3. Representación de algoritmos

3.1 Pseudocódigo

Ejemplo (factorial):

INICIO

LEER(n)

factorial = 1

PARA i DESDE 1 HASTA n HACER

factorial = factorial * i

FIN PARA

ESCRIBIR(factorial)

FIN

Ventajas: lenguaje independiente, fácil de entender y depurar.

3.2 Diagramas de flujo

- Símbolos clave:
 - Óvalo: Inicio/Fin.
 - Rombo: Decisión.
- Herramientas: Draw.io, Lucidchart.

3.3 Notaciones tabulares

- Tablas de decisión para visualizar reglas complejas.

3.4 Diagramas Nassi-Shneiderman

- Representación estructurada sin flechas.
- Más compactos que los flujogramas.

3.5 Lenguajes de especificación formal

- UML: Diagramas de clases, secuencia.
- BPMN: Modelado de procesos de negocio.
- Herramientas: StarUML, PlantUML.

3.6 Técnicas modernas

- Notebooks interactivos: Jupyter, Google Colab.
- Visualización: PythonTutor.
- Generación de código automática: GitHub Copilot, Codex.

4. Metodología de diseño de algoritmos

4.1 Planteamiento del problema

Preguntas clave:

- ¿Qué entradas y salidas hay?
- ¿Hay restricciones de tiempo o memoria?

4.2 Representación de datos

- Arrays vs Listas: Elegir según el tipo de operaciones.
- Grafos: Para representar relaciones complejas.

4.3 Descripción de instrucciones

- Técnica TOP-DOWN: Descomponer el problema en subproblemas más simples.
 - Ejemplo: Algoritmo de ordenación → comparar → intercambiar.

4.4 Verificación y optimización

- Pruebas: Usar casos típicos y casos límite (por ejemplo, listas vacías).
- Optimización: Buscar reducir la complejidad (de $O(n^2)$ a $O(n \log n)$).

5. Técnicas avanzadas de diseño algorítmico

5.1 Programación dinámica

- Guardar resultados intermedios (memoización).

- Ejemplo clásico: Fibonacci con memorización.

5.2 Algoritmos voraces (Greedy)

- Útiles cuando una solución óptima local conduce a una global.
- Ejemplos: problema del cambio de monedas, algoritmo de Dijkstra.

5.3 Backtracking

- Resolver problemas probando todas las combinaciones posibles.
- Aplicaciones: Sudoku, problema de las N reinas.
- Optimización: poda del árbol de búsqueda.

5.4 Divide y vencerás

- Pasos: dividir → resolver recursivamente → combinar.
- Ejemplo: Mergesort, Quicksort.

5.5 Algoritmos evolutivos

- Basados en evolución biológica (genéticos).
- Usan selección, cruce y mutación para resolver problemas complejos.

6. Eficiencia algorítmica

6.1 Análisis de complejidad

Complejidad	Ejemplo	Uso común
$O(1)$	Acceso a array	Tablas hash
$O(\log n)$	Búsqueda binaria	Árboles balanceados
$O(n \log n)$	Mergesort, Heapsort	Ordenación eficiente
$O(n^2)$	Bubble sort	Algoritmos simples, bucles
$O(n!)$	Fuerza bruta	Problemas combinatorios

7. Aplicaciones prácticas

7.1 Reutilización de algoritmos

- Patrones comunes reutilizables: búsqueda binaria, ordenación, etc.

7.2 Patrones de diseño

- Singleton: Una única instancia del objeto.
- Strategy: Intercambiar algoritmos en tiempo de ejecución.

7.3 Casos de estudio

- IA: K-Means (clustering), backpropagation.
- Big Data: MapReduce.
- Ciberseguridad: RSA, blockchain.
- Sistemas distribuidos: Paxos.
- Hardware moderno: uso de GPUs, computación cuántica (algoritmo de Grover).

8. Conclusiones

8.1 Comparativa de técnicas

Técnica	Complejidad típica	Uso ideal
Fuerza bruta	$O(n!)$	Problemas pequeños y simples
Divide y vencerás	$O(n \log n)$	Ordenación, búsqueda
Programación dinámica	$O(n^2)$	Subestructuras óptimas, mochila

8.2 Tendencias futuras

- Algoritmos cuánticos: búsqueda (Grover), factorización (Shor).
- Automatización de algoritmos mediante IA: generación de código (Copilot, GPT).

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. Constituye el puente entre el pensamiento lógico del ser humano y la ejecución automática en sistemas computacionales.

2. Elementos básicos de un lenguaje de programación

♦ Sintaxis y semántica

- **Sintaxis:** reglas que definen cómo deben escribirse las instrucciones.
- **Semántica:** significado lógico o funcional de las instrucciones escritas.

♦ Estructuras de control

- **Secuenciales:** ejecución línea a línea.
- **Condicionales:** `if`, `else`, `switch`.
- **Iterativas:** `for`, `while`, `do while`.

♦ Tipos de datos

- **Primitivos:** `int`, `char`, `float`, `bool`.
- **Estructurados:** arrays, registros.
- **Abstractos:** listas, pilas, colas, árboles.

♦ Tipado

- **Estático vs. Dinámico:** definidos en compilación o en ejecución.
- **Fuerte vs. Débil:** restricción o permisividad en la conversión de tipos.

♦ Otras cualidades destacables

- Legibilidad
- Modularidad
- Portabilidad
- Seguridad
- Eficiencia
- Facilidad de mantenimiento

3. Paradigmas de programación

Un **paradigma** es un modelo de pensamiento para organizar y resolver problemas a través del código. Cada uno propone una forma diferente de estructurar las instrucciones.

♦ Imperativo

- **Enfoque:** instrucción paso a paso, con cambios de estado.
- **Lenguaje típico:** C

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

♦ Declarativo

- **Enfoque:** se especifica qué se desea lograr, sin detallar cómo hacerlo.
- **Lenguaje típico:** SQL

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

♦ Funcional

- **Enfoque:** uso de funciones puras, sin efectos secundarios.
- **Lenguaje típico:** Haskell

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

♦ Lógico

- **Enfoque:** inferencia mediante hechos y reglas.
- **Lenguaje típico:** Prolog

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

♦ Orientado a objetos (OOP)

- **Enfoque:** encapsula datos y comportamiento en objetos.
- **Lenguaje típico:** Java

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

}

◆ Reactivo

- **Enfoque:** responde a eventos o flujos de datos de forma automática.
- **Lenguaje/framework típico:** Vue.js, RxJS

◆ Tiempo real

- **Enfoque:** respuesta inmediata ante estímulos del entorno.
- **Lenguaje típico:** C para sistemas embebidos

◆ Cuántico

- **Enfoque:** usa qubits, superposición y entrelazamiento.
- **Lenguaje típico:** Q#

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

4. Clasificación de lenguajes

◆ Por nivel de abstracción

- **Bajo nivel:** ensamblador
- **Medio nivel:** C, Rust
- **Alto nivel:** Python, Java

◆ Por forma de ejecución

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

◆ Por generación tecnológica

- **Clásicos:** Fortran, Pascal
- **Modernos:** Go, Kotlin, Swift

- **Emergentes:** Q#, Cirq

5. Herramientas y entornos de desarrollo

♦ Procesadores

- **Compiladores:** gcc, javac
- **Intérpretes:** python, node
- **Ensambladores:** NASM, MASM

♦ IDEs (Entornos de Desarrollo Integrado)

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

6. Aplicaciones comunes según lenguaje

Lenguaje	Aplicación típica
C/C++	Sistemas operativos, drivers, embebidos
Java	Aplicaciones Android, empresariales
Python	Inteligencia artificial, scripts, ciencia de datos
JavaScript/TS	Web (frontend y backend con Node.js)
Q#	Computación cuántica
SQL	Bases de datos relacionales
No-code/Low-code	Prototipado rápido, automatización de flujos

7. Tendencias actuales en programación

Estas corrientes reflejan hacia dónde se dirige el mundo del desarrollo de software:

- **Desarrollo en la nube (Cloud Computing):** ejecutar y escalar apps sin infraestructura local.
- **Inteligencia artificial y machine learning:** uso extensivo de Python y frameworks como TensorFlow o PyTorch.
- **Low-code/No-code:** herramientas visuales para construir apps sin escribir código (ej. Softr, glideapps).
- **Programación cuántica:** se inicia en entornos simulados (Azure Quantum, IBM Q Experience) con lenguajes como Q# o Cirq.
- **Entornos colaborativos online:** GitHub Codespaces, Replit, VS Code Live Share.

Propuesta didáctica – Actividad: “Viaje entre Lenguajes y Paradigmas”

Módulo: Programación

Curso: 1.º CFGS Desarrollo de Aplicaciones Multiplataforma

Duración estimada: 3 sesiones de 50 minutos

Objetivo de la actividad

Comprender de forma práctica los diferentes paradigmas de programación mediante la investigación, el desarrollo y la exposición de ejemplos funcionales en diversos lenguajes de programación.

Contenidos trabajados

- Paradigmas de programación: imperativo, declarativo, funcional, lógico, orientado a objetos, cuántico.
- Sintaxis y semántica.
- Tipado de datos.
- Herramientas de desarrollo: IDEs, compiladores, intérpretes.

Metodología

- Explicación previa por parte del docente con ejemplos en pseudocódigo y presentación visual de los paradigmas.
- Trabajo cooperativo en grupos.
- Aprendizaje activo a través de investigación, síntesis, programación y exposición.
- Uso de guías estructuradas, rúbricas y plantillas de apoyo.

Desarrollo de la actividad

1. Formación de grupos y asignación de un lenguaje representativo de un paradigma.
2. Investigación técnica del lenguaje y elaboración de una ficha técnica común.
3. Desarrollo de un ejemplo funcional y comentado en el lenguaje asignado.
4. Exposición oral breve (5 minutos por grupo) explicando el paradigma, el lenguaje y el ejemplo de código.

Evaluación (máximo 10 puntos)

La evaluación de la actividad se basará en cuatro aspectos clave:

- **Ficha técnica** (hasta 2 puntos): se valorará que esté completa, bien estructurada, con información clara sobre el lenguaje, su paradigma, sintaxis básica y uso principal.
- **Código funcional** (hasta 4 puntos): se evaluará que el código esté correctamente escrito, que funcione según lo esperado y que incluya comentarios explicativos que reflejen la comprensión del paradigma trabajado.
- **Presentación oral** (hasta 2 puntos): se tendrá en cuenta la claridad de la exposición, la capacidad para explicar el lenguaje y el código de forma comprensible, y el uso adecuado del tiempo asignado.

- **Participación, colaboración y actitud** (hasta 2 puntos): se valorará el trabajo en equipo, la implicación activa en la tarea, el respeto de los turnos de palabra y la actitud positiva durante todo el proceso.

Atención a la diversidad

- Refuerzo: guías paso a paso, uso de pseudocódigo y ejemplos resueltos.
- Ampliación: trabajo con más de un paradigma o lenguaje mixto.
- Adaptación: tiempos flexibles, apoyo visual y técnico individualizado, uso de recursos accesibles.

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

1. Introducción a la Programación Estructurada

• 1.1. Definición y Principios Básicos

- Paradigma de la programación estructurada: organización del flujo de control mediante estructuras bien definidas.
- Principales componentes: secuencia, selección e iteración.
- Relación con otros paradigmas: funcional, lógico, orientado a objetos.

• 1.2. Historia y Evolución

- Origen en la década de 1970 con Edgar Dijkstra y su crítica al spaghetti code.
- Aportaciones clave: modularización, simplificación, y enfoque en el control de flujo.

• 1.3. Objetivos del Uso de la Programación Estructurada

- Mejora de la legibilidad y mantenibilidad.
- Reducción de errores mediante control claro de flujo.
- Facilita la colaboración en proyectos grandes.

• 1.4. Ventajas

- Facilitación de la depuración.
- Reducción de complejidad y mayor facilidad de pruebas unitarias.
- Reutilización de código mediante funciones y procedimientos.
- Modularización y mantenimiento a largo plazo.

2. Estructuras Básicas

• 2.1. Secuencial

- Ejecución del código línea por línea.
- Ejemplo práctico: secuencia de asignaciones y operaciones.
- Importancia de las sentencias de inicialización.

• 2.2. Selección (Alternativas)

- If-Else: Comparación, condiciones de igualdad y desigualdad.
- Switch-Case: Alternativas múltiples y su eficiencia en comparación con if-else anidados.
- Operador Ternario: Optimización en expresiones condicionales simples.
- Ejemplos de aplicaciones: validación de entradas, decisiones de flujo.

• 2.3. Iteración (Bucles)

- For: Iteración definida, iteraciones en secuencias conocidas.
- While: Uso en condiciones que no se conocen de antemano.
- Do-While: Garantizar al menos una ejecución.
- Break y Continue: Control de flujo en bucles y optimización.
- Ejemplos de uso: navegación en estructuras de datos, procesamiento de archivos.

3. Funciones y Procedimientos

- **3.1. Diferencias Esenciales**

- Funciones: Valor de retorno, propiedades de inmutabilidad.
- Procedimientos: Acciones o efectos secundarios sin valor de retorno.
- Comparativa: Modularidad en funciones y procedimientos en sistemas grandes.

- **3.2. Parámetros y su Manejo**

- Por valor: Cópia de la información.
- Por referencia: Paso directo del valor, modificación de la variable original.
- Parámetros por defecto: Evitar sobrecarga de funciones.

- **3.3. Ámbito de las Variables**

- Variables locales y globales: prácticas recomendadas y desventajas de usar variables globales.
- Ámbito estático: Variables que persisten durante la ejecución del programa.
- Accesibilidad y riesgos asociados con la manipulación incorrecta del ámbito.

- **3.4. Valores de Retorno**

- Tipos básicos y complejos.
- Validación de valores de retorno y su uso para manejar flujos de control.

- **3.5. Recursividad Avanzada**

- Fundamentos de la recursividad: caso base y caso recursivo.
- Optimización de recursividad: Tail recursion, optimización en lenguajes modernos.
- Desventajas: Peligro de desbordamientos de pila, ineficiencia en algunos casos.

- **3.6. Ejemplos de Recursividad Completa**

- Ejemplo práctico de Fibonacci, recorrido en árboles binarios.
- Análisis de complejidad recursiva.

4. Control de Flujo Avanzado

- **4.1. Manejo de Excepciones**

- Introducción a la gestión de errores: diferencia entre errores y excepciones.
- Try-Catch: Manejo eficiente de excepciones.
- Finally: Liberación de recursos y gestión de errores críticos.
- Ejemplo de manejo de excepciones en operaciones de E/S.

- **4.2. Sentencias de Control Adicionales**

- Goto: Casos de uso controvertidos y mejores alternativas.
- Assert: Evaluación de condiciones en tiempo de ejecución para asegurar el comportamiento esperado.

- **4.3. Excepciones Personalizadas**

- Creación y manejo de excepciones propias en lenguajes orientados a objetos.

5. Optimización y Eficiencia en Programación Estructurada

- **5.1. Análisis de Complejidad Algorítmica**

- Notación Big-O: Análisis de la eficiencia de algoritmos iterativos y recursivos.
- Comparación entre algoritmos recursivos e iterativos.
- Ejemplo de comparación: Búsqueda binaria vs. búsqueda lineal.

- **5.2. Optimización de Recursos**
 - Uso eficiente de la memoria: pilas, colas y estructuras dinámicas.
 - Gestión de memoria estática y dinámica.
 - Técnicas de reducción de ciclos de CPU: Bucles ineficientes, operaciones costosas.
- **5.3. Técnicas Avanzadas de Optimización**
 - Memorización: Optimización en algoritmos recursivos (Fibonacci, cálculos matemáticos).
 - Caching: Almacenamiento en caché para mejorar el rendimiento.

6. Comparación: Programación Estructurada vs. Programación Orientada a Objetos

- **6.1. Enfoque de Diseño**
 - Programación estructurada: basarse en funciones y procedimientos.
 - Programación orientada a objetos: centrado en objetos, clases y herencia.
- **6.2. Manejo de Datos**
 - PE: manipulación directa de variables y estructuras de datos.
 - POO: encapsulamiento, ocultación de datos y responsabilidad de los objetos.
- **6.3. Escalabilidad y Mantenibilidad**
 - PE: Ideal para proyectos pequeños a medianos.
 - POO: Escalabilidad para proyectos grandes y complejos.
- **6.4. Ventajas y Desventajas**
 - Comparativa práctica entre ambos enfoques según el contexto de uso.

7. Tendencias Modernas en Programación Estructurada

- **7.1. Lenguajes Modernos con Programación Estructurada**
 - Rust: Lenguaje de sistemas que incorpora PE y paradigmas funcionales.
 - Go: Lenguaje sencillo y eficiente, optimizado para concurrencia.
 - Python y TypeScript: Uso moderno de PE con type hints y tipos estáticos.
- **7.2. Paradigmas Híbridos**
 - Uso de programación estructurada dentro de lenguajes orientados a objetos.
 - PE y programación funcional: Lenguajes como Haskell, Scala, y su integración con PE.
- **7.3. Buenas Prácticas**
 - Uso de funciones pequeñas (<30 líneas), modularización, y documentación clara.
 - Evitar el uso de variables globales y gestión adecuada de excepciones.
 - Padrón de diseño de software: Técnicas de estructuración como MVC o capas.

8. Casos Prácticos y Aplicaciones

- **8.1. Implementación de Algoritmos**
 - Ejemplo práctico de un algoritmo de ordenación (como QuickSort o MergeSort) usando PE.
 - Análisis de recursividad en estructuras de árboles binarios de búsqueda.
- **8.2. Análisis y Optimización de Código**
 - Evaluación de código ineficiente y mejora a través de técnicas estructuradas.
 - Refactorización de código: Optimización sin alterar la funcionalidad.

9. Conclusiones

- **9.1. Relevancia en el Mundo Moderno**

- La programación estructurada sigue siendo fundamental en la formación de programadores y es clave para proyectos de pequeña y mediana escala.

- **9.2. Adaptabilidad y Flexibilidad**

- Aunque la programación orientada a objetos ha ganado popularidad, el enfoque estructurado sigue siendo esencial para ciertos tipos de sistemas y es un pilar fundamental para aprender otros paradigmas.

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

1. Introducción

- Definición: La programación modular es una técnica de diseño de software que divide un programa en módulos autónomos. Cada módulo es responsable de una tarea específica y se comunica con otros módulos a través de interfaces bien definidas.
- Objetivos:
 - Mejorar la mantenibilidad: Facilitar cambios sin afectar todo el sistema.
 - Fomentar la reutilización de código: Los módulos pueden ser reutilizados en diferentes proyectos.
 - Promover un diseño escalable: Facilita la expansión y modificación de sistemas complejos.

2. Fundamentos de la Programación Modular

2.1. ¿Qué es un Módulo?

- Un módulo es una unidad funcional de código que encapsula una tarea específica.
- Características:
 - Cohesión alta: Cada módulo tiene una responsabilidad única y clara.
 - Acoplamiento bajo: Los módulos tienen dependencias mínimas entre sí, lo que reduce el impacto de los cambios.
 - Interfaz clara: La entrada y la salida de los módulos están bien definidas, facilitando su interacción.

2.2. Cohesión y Acoplamiento

- Cohesión: Se refiere a cuán relacionadas están las funcionalidades dentro de un mismo módulo.
 - Alta cohesión: El módulo realiza una tarea bien definida.
 - Baja cohesión: El módulo tiene muchas responsabilidades y tareas no relacionadas.
 - Ejemplo: Un módulo de gestión de usuarios tiene alta cohesión si solo se encarga de funciones relacionadas con usuarios (registrar, eliminar, modificar usuarios).
- Acoplamiento: Se refiere a cómo interactúan los módulos entre sí. El objetivo es mantener acoplamiento bajo, es decir, minimizar las dependencias entre módulos.
 - Acoplamiento bajo: Los módulos interactúan solo a través de interfaces claras.
 - Acoplamiento alto: Los módulos dependen fuertemente unos de otros.
 - Ejemplo: Si un módulo de autenticación depende directamente de un módulo de base de datos, se genera un alto acoplamiento. En su lugar, deberían interactuar a través de una interfaz definida.

2.3. Circulación de Datos entre Módulo

- Los módulos se comunican mediante parámetros de entrada/salida o intercambio de mensajes.
 - Parámetros: Los datos se pasan directamente entre funciones (ej: funciones con parámetros de entrada y salida).

- Mensajes: Los módulos envían mensajes para intercambiar información (ej: en sistemas basados en eventos).
- Interfaz: Cada módulo debe exponer una interfaz clara que defina qué datos acepta y qué datos devuelve, sin depender de detalles internos del módulo.

3. Diseño de Funciones

3.1. Principios Básicos

- Principio SOLID: Cada función debe tener una única responsabilidad.
- Nombres claros: Usa nombres descriptivos para las funciones.
- Parámetros limitados: Idealmente, las funciones deben recibir un máximo de 3 parámetros. Si se superan, considerar el uso de objetos.

3.2. Uso de Variables Globales

- Variables Globales: Son accesibles desde cualquier parte del programa, pero pueden crear problemas al generar efectos colaterales no deseados.
 - Problemas: Confusión en el flujo de datos y dificultad para hacer pruebas.
 - Alternativa: Usar variables locales o objetos para encapsular el estado.
- Solución moderna: Gestión de Estado (ej: Vuex):
 - En aplicaciones modernas, como las basadas en Vue.js, Redux (para React) o Vuex, se usa un gestor de estado global para almacenar el estado compartido entre componentes sin necesidad de usar variables globales directas.
 - Vuex: Permite gestionar el estado de la aplicación de manera centralizada y predecible, asegurando que los cambios se realicen de forma controlada.

3.3. Comunicación entre Componentes y Módulos

- Componentes: En arquitecturas modernas, los componentes pueden ser módulos autónomos que gestionan su propia interfaz de usuario y lógica.
 - Comunicación entre componentes: En un sistema modular, los componentes deben ser independientes, pero aún así deben comunicarse entre sí.
 - Métodos:
 - Props y Eventos: En sistemas como Vue.js o React, la comunicación entre componentes se maneja a través de props (pasando datos de padre a hijo) y eventos (de hijo a padre).
 - Eventos Globales o Store: En aplicaciones más complejas, se puede usar un gestor de estado global como Vuex o Redux, que permite que los componentes se sincronicen sin necesidad de pasar props o eventos manualmente entre ellos.

4. Recursividad

4.1. Estructura Básica de la Recursividad

- Caso Base: Condición de salida de la recursión (ej: if $n == 0$: return 1).
- Caso Recursivo: Llamada recursiva con el problema reducido (ej: return $n * \text{factorial}(n-1)$).

4.2. Optimización de la Recursividad

- Memorization: Almacenar los resultados de llamadas recursivas para evitar cálculos repetidos.
- Tail Recursion: Optimización de la recursión en la cola, permitiendo que el compilador optimice la llamada recursiva.

4.3. Stack Overflow

- Si la recursividad no tiene un caso base o no reduce correctamente el problema, se puede producir un desbordamiento de pila.

5. Librerías y Gestión de Componentes

5.1. Tipos de Librerías

- Librerías Estándar: Vienen integradas con el lenguaje (ej: math, datetime en Python).
- Librerías Externas: Se instalan por separado (ej: numpy, requests).

5.2. Buenas Prácticas

- Documentación: Asegúrate de incluir docstrings y archivos README.md.
- Versionado: Utiliza Semantic Versioning para mantener compatibilidad entre versiones.

6. Modularidad en Sistemas Complejos

6.1. Modularidad en Arquitecturas de Componentes

- Microservicios: Sistemas basados en servicios pequeños e independientes que se comunican entre sí. Cada microservicio tiene su propia base de datos y lógica.
- Capas Modulares: Los sistemas pueden dividirse en capas, cada una con una responsabilidad distinta (Ej: Capa de presentación, Capa de negocio, Capa de datos).

6.2. Comunicación entre Módulos

- Interfaz de Comunicación: Los módulos se comunican entre sí a través de APIs (REST, gRPC, etc.).
 - RESTful APIs: Permiten la interacción entre diferentes módulos de forma estándar.
 - Mensajes/Eventos: Los módulos pueden intercambiar datos mediante mensajes (por ejemplo, en arquitecturas basadas en eventos).

6.3. Modularidad en el Frontend

- Estado Global: Para mantener el estado de la aplicación entre componentes, se puede usar un store como Vuex o Redux, gestionando el estado centralizado sin recurrir a variables globales.
 - Ejemplo (Vuex): Los módulos en Vuex pueden representar secciones independientes de la aplicación, como el usuario, autenticación o configuración.

7. Conclusión

- La programación modular es fundamental para construir sistemas escalables, mantenibles y flexibles. Facilita la colaboración entre equipos y la reutilización de código, lo que permite crear aplicaciones robustas y de fácil mantenimiento.

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

1. Introducción

1.1. Evolución Histórica

- **Origen:**
 - Programación estructurada (años 60-70): lenguajes como *C* y *Pascal*.
 - **Limitaciones:** Rigidez, dificultad para modelar problemas complejos.
- **Surgimiento de POO** (años 80):
 - *Smalltalk* (primer lenguaje puro orientado a objetos).
 - Lenguajes híbridos: *C++* (extensión de *C*), *Objective-C*.
 - Consolidación: *Java*, *C#*, *Kotlin* (POO pura + seguridad tipográfica).
 - Influencia en multiparadigma: *Python*, *TypeScript* (flexibilidad + POO).

1.2. Problemas del Paradigma Estructurado

- **Modelado:** Dificultad para representar entidades reales (ej: "Persona" con datos + comportamientos).
- **Abstracción:** Código repetitivo, poca reutilización.
- **Acoplamiento:** Funciones y datos separados → cambios cascada.
- **Mantenimiento:** Caos en sistemas grandes (ej: miles de líneas en *C*).

1.3. Ventajas Clave de la POO

- **4 Pilares Fundamentales:**
 - **Abstracción:** Clases como moldes de objetos del mundo real (ej: *Clase Coche* con atributos y métodos).
 - **Encapsulamiento:** Protección de datos (ej: *private* en Java) → Evita accesos no controlados.
 - **Herencia:** Jerarquía y reutilización (ej: *Clase Vehículo* → *Coche*, *Moto*).
 - **Polimorfismo:** Mismo método, múltiples formas (ej: *Animal.sonido()* → *Perro.ladra()*, *Gato.maula()*).
- **Beneficios Prácticos:**
 - Código modular (divide y vencerás).
 - Escalabilidad: Más intuitivo para equipos grandes.
 - Testing: Aislamiento de componentes.
-

2. Fundamentos Teóricos de la POO

2.1. Conceptos Básicos

Clase vs Objeto:

- **Clase:** "Molde" abstracto (ej: `class Coche` en Java/Python).
- **Objeto:** Instancia concreta (ej: `Coche miTesla = new Coche()`).
Diferencia clave: La clase define *qué* es, el objeto es *uno específico*.

Atributos:

- *De instancia:* Únicos por objeto (ej: `color`).
- *Estáticos:* Compartidos por todas las instancias (ej: `contadorCoches`).
- *Constantes:* Valores fijos (ej: `final PI = 3.14` en Java).

Métodos:

- *De instancia:* Actúan sobre un objeto (`acelerar()`).
- *Estáticos:* No requieren instancia (`Coche.getTotalCoches()`).

Visibilidad:

- `public`: Acceso desde cualquier clase.
 - `private`: Solo desde la propia clase (encapsulamiento).
 - `protected`: Acceso desde clases hijas.
-

2.2. Encapsulamiento y Control de Acceso

Objetivo: Proteger el estado interno (ej: evitar `miCoche.velocidad = -100`).

Mecanismos:

- *Getters/Setters:* Validación al acceder/modificar (ej: `setVelocidad(int v) { if (v >= 0) this.velocidad = v; }`).
- *Propiedades automáticas:* Simplifican código (ej: `public int Velocidad { get; set; }` en C#).

Ventajas:


- Seguridad: Control sobre cambios.
 - Flexibilidad: Cambiar implementación interna sin afectar otras clases.
-

2.3. Relaciones entre Clases



Tipos de relaciones:

1. **Asociación:** Uso temporal (ej: **Profesor** usa **Pizarra**).
2. **Agregación:** "Tiene-pero-no-esencial" (ej: **Universidad** tiene **Estudiantes**, pero estos existen sin ella).
3. **Composición:** "Es-esencial" (ej: **Casa** tiene **Habitaciones**; si la casa se destruye, las habitaciones también).

Dependencia:

-  **Bajo Acoplamiento (o "No pegues todo con superpegamento")**
 - **Problema:** Si una clase **depende mucho** de otra (ej: **Jugador** crea su propia **Arma** con `new Arma()`), es como soldarlas: si cambias una, se rompe la otra.
 - **Solución: Inyección de Dependencias (DI)** → Alguien *externo* (un "contenedor") le pasa las herramientas al **Jugador**, en lugar de que él las fabrique.
- **Patrones clave:**
 - **IoC (Inversión de Control):** El framework gestiona objetos.
 - **DI Containers:** Spring, Unity (evitan `new Servicio()` manual).

Los 2 Conceptos Clave:

1. **IoC (Inversión de Control):**
 - *Normal:* Tú controlas todo (`new Arma()`).
 - *IoC:* El **framework** (como Spring) controla la creación de objetos.
 -  **Analogía:**
 - Sin IoC: Tú cocinas todo desde cero.
 - Con IoC: Pides comida a domicilio (Spring es el repartidor).
2. **Contenedores DI (Spring, Unity):**
 - Son "cajas mágicas" que **guardan objetos listos para usar**.
 - Tú dices: "Necesito un **Arma**", y el contenedor te la da *ya creada*.
 -  **Ejemplo:** Unity (para videojuegos) gestiona **Armas**, **Enemigos**, etc., sin que tú los instances manualmente.

3. Pilares de la Programación Orientada a Objetos

3.1. Abstracción e interfaces

- Diferencia entre clase abstracta e interface: cuándo usar cada una.
- Ejemplos en Java (abstract class, interface), C# (interface, abstract) y Python (ABC, @abstractmethod).
- Interfaces funcionales (Java 8 y lambdas).
- Mixins y herencia múltiple simulada (Python).

3.2. Herencia

- Tipos de herencia:
 - Simple, multinivel, jerárquica.

- Múltiple: presente en C++, con resolución del problema del diamante mediante virtual inheritance.
- Sobrescritura de métodos (@Override en Java).
- Clases finales (Java: final, C#: sealed).
- Uso y abuso de la herencia: por qué preferir la composición en muchos casos.

3.3. Polimorfismo

- Polimorfismo de sobrecarga: métodos con el mismo nombre y diferentes firmas.
- Polimorfismo de sobrescritura: redefinición del comportamiento en clases hijas.

Enlace dinámico y comportamiento en tiempo de ejecución.

Ejemplo:

Animal a = new Perro();

a.hacerSonido(); // Ejecuta el método de Perro

- Polimorfismo paramétrico: Generics en Java (List<T>), C# (List<T>).

3.4. Principios SOLID (diseño orientado a objetos de calidad)

- S: Single Responsibility – una clase debe tener una única responsabilidad.
- O: Open/Closed – abierta a extensión, cerrada a modificación.
- L: Liskov Substitution – las subclases deben poder sustituir a la clase padre.
- I: Interface Segregation – muchas interfaces específicas mejor que una general.
- D: Dependency Inversion – depender de abstracciones, no de implementaciones.
- Ejemplos prácticos en Java, C# o pseudocódigo.

4. Lenguajes de Programación Orientados a Objetos

4.1. Comparativa de características

Lenguaje	POO pura	Herencia múltiple	Gestión de memoria	Casos de uso
Java	Sí	No (interfaces)	Automática (GC)	Web, Android
C++	No	Sí	Manual / RAII	Juegos, sistemas
Python	No	Sí (mixins)	Automática (GC)	IA, scripting
C#	Sí	No (interfaces)	Automática (GC)	.NET apps, desktop

4.2. Casos de estudio

- Java: Spring Framework – IoC, Beans, Repositorios.
- C#: Entity Framework – ORM orientado a objetos, LINQ.
- Python: Django – modelos como clases, ORM, vistas como métodos.

5. Diseño Avanzado y Patrones Orientados a Objetos

5.1. Patrones de diseño OO

- Creacionales:
 - Factory Method, Abstract Factory, Singleton (problemas con hilos).
- Estructurales:
 - Adapter, Decorator, Composite.
 - Ejemplo: Decorator en flujos de entrada/salida en Java.
- Comportamiento:
 - Observer (eventos), Strategy, Command, State.

- Beneficios: reutilización, mantenibilidad, bajo acoplamiento.

5.2. Arquitecturas basadas en POO

- MVC (Modelo-Vista-Controlador): separación de responsabilidades.
- Clean Architecture / Hexagonal Architecture: dominio en el centro, independencia de frameworks.
- Aplicación en microservicios, APIs REST, frameworks modernos.

6. Aplicaciones prácticas

6.1. Ejercicios y ejemplos resueltos

- Jerarquía de figuras geométricas (clases abstractas, métodos polimórficos).
- Sistema de biblioteca: herencia entre Libro, Revista, etc.
- Simulación de un sistema bancario con encapsulamiento y clases auxiliares.

6.2. Errores frecuentes y buenas prácticas

- Abuso de herencia (violación del principio de responsabilidad única).
- Clases "Dios" o monolíticas: exceso de responsabilidades.
- Acoplamiento excesivo entre clases.
- Recomendaciones: preferir composición, aplicar principios SOLID, usar interfaces.

7. Tendencias actuales y conclusiones

7.1. Evolución y tendencias de la POO

- Convergencia con programación funcional (Kotlin, Scala, F#).
- Aplicación en microservicios, contenedores (Docker, Kubernetes).
- Orientación a objetos reactiva (RxJava, ReactiveX).
- Frameworks híbridos: React con TypeScript y patrones OO.

7.2. Conclusión

- La orientación a objetos continúa siendo la base de la ingeniería de software moderna.
- Su combinación con otros paradigmas permite soluciones más robustas, mantenibles y adaptables.
- El dominio de la POO es esencial para el desarrollo de aplicaciones escalables, limpias y seguras.

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. Orígenes e impacto

- Creado por Dennis Ritchie en 1972 en los laboratorios Bell como evolución de B y BCPL.
- Desarrollado inicialmente para reescribir el sistema operativo UNIX.
- Base de lenguajes como C++, Java, Objective-C, Rust.
- Estandarizado por ANSI (C89) y posteriormente ISO (C99, C11, C17).

1.2. Características fundamentales

- Lenguaje de medio nivel: combina abstracciones de alto nivel con acceso a bajo nivel (punteros, memoria, registros).
- Portabilidad: fácil adaptación entre plataformas (sistemas embebidos, UNIX/Linux, microcontroladores).
- Eficiencia y rendimiento: compilación directa a código máquina sin máquina virtual.
- Estructurado y modular, aunque no orientado a objetos nativamente.
- Sintaxis compacta y expresiva, con solo 32 palabras clave en ANSI C.

1.3. Ventajas y limitaciones

Ventajas	Limitaciones
Control total del hardware	Gestión manual de memoria
Gran comunidad y documentación	No posee tipado seguro por defecto
Portabilidad y compatibilidad	Sin soporte nativo para programación OOP
Base de sistemas operativos	Menor abstracción para grandes proyectos

2. Elementos del Lenguaje

2.1. Estructura general de un programa

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

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


Componentes clave:

- Directivas del preprocesador: #include, #define, #ifdef.
- Función principal int main() como punto de entrada.
- Sentencias terminadas en ;.
- Comentarios: // (C99), /* ... */ (todas las versiones).

2.2. Tipos de datos y variables

- Primitivos: char, int, float, double.
- Modificadores: short, long, unsigned, signed.
- Cadenas: arrays de char terminados en \0.

Ejemplo:

```
unsigned long int contador = 0;
```

2.3. Operadores

Tipo	Ejemplos
Aritméticos	+, -, *, /, %
Relacionales	==, !=, <, >
Lógicos	&&, ^
Asignación	=, +=, *=
Bit a bit	&, ^

3. Estructura Modular de un Programa C

3.1. Separación de responsabilidades

- Archivos .h: declaración de constantes, tipos, funciones.
- Archivos .c: implementación de funciones.
- Uso de extern para variables globales compartidas entre módulos.

3.2. Preprocesador

- Macros (#define PI 3.14)
- Inclusión condicional:

```
#ifndef __MI_LIBRERIA_H__  
#define __MI_LIBRERIA_H__  
// contenido  
#endif
```

- Uso avanzado: #pragma, #error, #undef.

4. Funciones: Librería estándar y definidas por el usuario

4.1. Funciones estándar

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

<string.h>	strlen(), strcpy(), strcmp()
<math.h>	sqrt(), pow(), sin()

Uso:

```
#include <math.h>
```

```
double r = sqrt(16.0);
```

4.2. Funciones de usuario

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

- Prototipos declarados en .h.
- Paso de parámetros por valor y por referencia (punteros).
- Recursividad:

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

5. Entorno de Compilación

5.1. Fases del proceso

1. Preprocesado → expansión de macros (archivo.i)
2. Compilación → conversión a código objeto (archivo.o)
3. Enlazado (linking) → unión de objetos y librerías
4. Ejecución → se genera binario (.exe, .out)

5.2. Compiladores y herramientas

- GCC (GNU Compiler Collection):

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

- Clang (LLVM): compilación rápida, análisis estático.
- Make y Makefile:

```
programa: main.o lib.o
```

```
gcc -o programa main.o lib.o
```

6. Herramientas para Elaboración y Depuración

6.1. Depuración con GDB

```
gdb ./programa
```

```
(gdb) break main
```

```
(gdb) run
```

```
(gdb) next
```

```
(gdb) print variable
```

6.2. Detección de errores

- Valgrind: gestión de memoria.

```
valgrind ./programa
```

- Sanitizers (GCC/Clang):

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

6.3. Optimización

- Niveles: -O0, -O1, -O2, -O3, -Os.
- Profiling con gprof:

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

7. Proyecto Integrado: Ejemplo Realista en Módulos

Estructura del proyecto:

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

Contenido:

```
operaciones.h
int suma(int a, int b);
```

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

```
main.c
#include <stdio.h>
#include "operaciones.h"
```

```
int main() {
    printf("Suma: %d\n", suma(4, 5));
    return 0;
}
```

Compilación:

```
gcc -c operaciones.c
gcc -c main.c
gcc -o programa main.o operaciones.o
```

8. Conclusión

- Lenguaje clave en programación de sistemas, embebidos, compiladores y sistemas operativos.
- Su conocimiento es fundamental para comprender la estructura interna de programas.
- Pese a su edad, C se mantiene vigente por su eficiencia, control, portabilidad y cercanía al hardware.

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

1.1. Estructuras de datos en C

- Estáticas: tamaño fijo en tiempo de compilación (arrays, struct).
- Dinámicas: tamaño variable gestionado en tiempo de ejecución (listas, árboles, pilas, colas).
- Ventaja clave: control absoluto sobre el uso y la gestión de memoria.

1.2. Objetivos del tema

- Dominar estructuras dinámicas mediante punteros.
- Usar funciones de entrada/salida con eficacia.
- Aplicar punteros a funciones para lograr código modular, flexible y reutilizable.

2. Estructuras de Datos Estáticas

2.1. Arrays y estructuras (struct)

```
int numeros[10];
struct Persona {
    char nombre[50];
    int edad;
};
```

- Arrays: contenedores de tamaño fijo y homogéneo.
Structs: agrupan datos heterogéneos bajo un mismo identificador.

2.2. Inicialización y acceso

```
struct Persona p1 = {"Lucía", 30};
printf("%s", p1.nombre);
```

- Acceso directo a campos con ..
- Copias de structs completas: struct Persona p2 = p1;.

3. Estructuras de Datos Dinámicas

3.1. Gestión dinámica de memoria

Función	Descripción
malloc()	Reserva sin inicializar
calloc()	Reserva e inicializa a 0
realloc()	Redimensiona bloques ya reservados
free()	Libera memoria y evita fugas

Ejemplo:

```
int* v = malloc(5 * sizeof(int));
if (v != NULL) { /* uso seguro */ }
free(v);
```

3.2. Listas enlazadas

```
struct Nodo {
    int dato;
    struct Nodo* siguiente;
};
```

- Gestión manual de nodos, uso intensivo de punteros.
- Inserción al inicio:

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

3.3. Pilas y colas

- Pila (LIFO): usar inserción y extracción en la cabeza.
- Cola (FIFO): requiere control de cabeza y cola.
- Casos de uso: procesamiento de tareas, evaluadores de expresiones, navegación por árbol.

4. Entrada y Salida de Datos

4.1. Entrada estándar

Función	Uso
scanf()	Lectura formateada (cuidado con buffer overflow)
fgets()	Lectura segura de cadenas (limita tamaño)
getchar() / getch()	Lectura carácter a carácter

Ejemplo seguro:

```
char nombre[50];
fgets(nombre, sizeof(nombre), stdin);
```

4.2. Archivos

```
FILE* f = fopen("archivo.txt", "r");
if (f != NULL) {
    char buffer[100];
    while (fgets(buffer, sizeof(buffer), f)) {
        puts(buffer);
    }
    fclose(f);
}
```

- Apertura: "r", "w", "a", "rb"...
- Funciones clave: fopen, fprintf, fscanf, fgets, fclose.

5. Gestión de Punteros

5.1. Fundamentos

- Dirección: operador &.
- Indirección: operador *.
- Aritmética:

```
int arr[3] = {10, 20, 30};  
int* p = arr;  
printf("%d", *(p + 1)); // Output: 20
```

5.2. Punteros a estructuras

```
struct Persona* p = &p1;  
printf("%s", p->nombre);
```

5.3. Punteros dobles

- Manipular punteros desde funciones:

```
void asignar(int** p) {  
    *p = malloc(sizeof(int));  
}
```

- Uso en estructuras complejas: matrices dinámicas (int** matriz), listas enlazadas dobles, árboles.

6. Punteros a Funciones

6.1. Declaración y uso

```
int suma(int a, int b) { return a + b; }  
int (*pf)(int, int) = suma;  
printf("%d", pf(2, 3)); // 5
```

6.2. Aplicaciones

- Callbacks con qsort():

```
int comparar(const void* a, const void* b) {  
    return (*(int*)a - *(int*)b);  
}  
int v[] = {5, 2, 8};  
qsort(v, 3, sizeof(int), comparar);
```

- Diseño flexible: selección de funciones en tiempo de ejecución (ej. menú de opciones dinámico).

7. Ejemplo Integrado: Lista + Archivo

```
void guardar(struct Nodo* cabeza, const char* archivo) {  
    FILE* f = fopen(archivo, "w");  
    while (cabeza != NULL) {  
        fprintf(f, "%d\n", cabeza->dato);  
        cabeza = cabeza->siguiente;  
    }  
    fclose(f);  
}
```

- Muestra integración real de memoria dinámica, structs, punteros y E/S.

8. Herramientas y Buenas Prácticas

8.1. Herramientas

- Valgrind: detectar fugas y errores de acceso:

`valgrind --leak-check=full ./mi_programa`

- GDB: depuración paso a paso, inspección de punteros:

`gdb ./mi_programa`

`(gdb) print *puntero`

8.2. Buenas prácticas

- Verificar siempre el resultado de `malloc()`.
- Liberar memoria cuando ya no se necesita (`free()`).
- Documentar claramente qué función reserva y cuál libera la memoria.
- Evitar “dangling pointers” (punteros colgantes).

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

1. Introducción

1.1. Relevancia de la manipulación de datos en los sistemas actuales

- Elemento central en el funcionamiento de los SGBD.
- Manipulación eficiente = rendimiento + escalabilidad.
- Aplicación transversal: sistemas empresariales, móviles, web, IoT y Big Data.

1.2. Evolución histórica

- De los modelos jerárquicos y de red a bases de datos relacionales (años 70, modelo de Codd).
- Aparición de NoSQL ante necesidades de escalabilidad horizontal y flexibilidad (2000s).
- Emergen soluciones híbridas (NewSQL, multimodelo).

1.3. Comparativa general entre paradigmas

Característica	SQL (Relacional)	NoSQL
Modelo de datos	Tabular	Documentos, grafos, clave-valor, columnas
Consistencia	Alta (ACID)	Eventual (BASE)
Escalabilidad	Vertical	Horizontal
Flexibilidad del esquema	Rígido	Flexible/dinámico

2. Modelos de Datos

2.1. Modelo relacional (SQL)

- Representación basada en tablas (relaciones).
- Integridad mediante claves primarias, foráneas, restricciones.
- Normalización para evitar redundancias.

2.2. Modelos NoSQL

- Documental (MongoDB): documentos JSON/BSON.
- Clave-valor (Redis, DynamoDB): almacenamiento simple y ultra rápido.
- Columnares (Cassandra, HBase): útil en grandes volúmenes y análisis masivo.
- Grafos (Neo4j): relaciones complejas como ciudadanos-redes sociales.

2.3. Modelos híbridos: NewSQL y multimodelo

- Bases que combinan ACID con escalabilidad NoSQL (ej. CockroachDB, ArangoDB).
- Enfoque de “polyglot persistence”: elegir el modelo según el caso de uso.

3. Lenguajes de Manipulación de Datos

3.1. SQL: Lenguaje estándar en BBDD relacionales

- Declarativo, potente y ampliamente adoptado.
- Subdivisiones:
 - DML: INSERT, UPDATE, DELETE, SELECT.

- DDL: CREATE, ALTER, DROP.
- DCL y TCL: control de permisos y transacciones.

3.2. Lenguajes en NoSQL

- MongoDB Query Language (MQL): consultas sobre estructuras anidadas.
- Cassandra Query Language (CQL): inspirado en SQL, orientado a columnas.
- Cypher (Neo4j): sintaxis declarativa para grafos.

3.3. Diferencias conceptuales clave

- NoSQL prioriza flexibilidad, rendimiento y disponibilidad.
- SQL prioriza consistencia, estructura fija, y seguridad transaccional.

4. Manipulación de Datos en SQL

4.1. Operaciones básicas

- INSERT INTO tabla (...) VALUES (...)
- UPDATE tabla SET campo = valor WHERE condición
- DELETE FROM tabla WHERE condición

4.2. Consultas avanzadas

4.2.1. Funciones de agregación

- COUNT, SUM, AVG, MIN, MAX
- Agrupación con GROUP BY y filtrado con HAVING.

4.2.2. Subconsultas

- En cláusulas WHERE, SELECT, FROM.
- Subconsultas correlacionadas vs independientes.

4.2.3. Operadores de conjuntos

- UNION, INTERSECT, EXCEPT
- Combinación de resultados de varias consultas.

4.2.4. Joins

- INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN
- Comparación de rendimiento entre tipos de join y uso de índices.

4.3. Transacciones y control de concurrencia

- Propiedades ACID.
- Uso de BEGIN, COMMIT, ROLLBACK.
- Aislamiento: niveles (READ COMMITTED, SERIALIZABLE, etc.).
- Mecanismos: bloqueo optimista/pesimista, control multiversión (MVCC).

5. Manipulación de Datos en NoSQL

5.1. Caso de estudio: MongoDB

5.1.1. Inserción:

db.usuarios.insertOne({ nombre: "Ana", edad: 30 });

5.1.2. Actualización:

db.usuarios.updateOne({ nombre: "Ana" }, { \$set: { edad: 31 } });

5.1.3. Eliminación:

db.usuarios.deleteOne({ nombre: "Ana" });

5.1.4. Agregación:

- aggregate(): \$match, \$group, \$project, \$lookup.
- Consultas equivalentes a joins relacionales.

5.1.5. Comparativa SQL vs MongoDB

SQL	MongoDB
-----	---------

SELECT * FROM users	db.users.find()
INSERT INTO ...	insertOne()
UPDATE ...	updateOne()

5.2. Caso de estudio: Cassandra (CQL)

5.2.1. Inserción y actualización:

INSERT INTO clientes (id, nombre) VALUES (1, 'Juan');

UPDATE clientes SET nombre = 'Carlos' WHERE id = 1;

5.2.2. Consulta:

- Uso de WHERE en claves de partición.
- Limitaciones en joins y subconsultas.
- Operaciones eficientes si el esquema está bien diseñado.

6. Optimización de Consultas

6.1. En entornos relacionales (SQL)

6.1.1. Índices

- Tipos: B-Tree, Hash, Full-Text.
- Impacto positivo en WHERE, JOIN, ORDER BY.

6.1.2. Planificación de ejecución

- EXPLAIN, ANALYZE (PostgreSQL, MySQL)
- Identificación de cuellos de botella.

6.1.3. Reescritura de consultas

- Simplificar subconsultas.
- Eliminar operaciones costosas.
- Uso de vistas materializadas.

6.2. En bases NoSQL

6.2.1. Diseño del modelo según consulta

- “Query-first design”: diseñar el esquema en base a las consultas.
- Denormalización como técnica habitual.

6.2.2. Índices en MongoDB y Cassandra

- Índices compuestos, geoespaciales, TTL (time to live).
- Cuidado con el crecimiento excesivo de índices.

6.2.3. Técnicas de partición y replicación

- Sharding: fragmentación horizontal.
- Replicación: alta disponibilidad y tolerancia a fallos.

6.3. Herramientas y análisis de rendimiento

- PostgreSQL: EXPLAIN ANALYZE, pg_stat_statements.
- MongoDB: explain(), Atlas Performance Advisor.
- Cassandra: nodetool, tracing.

7. Configuración Avanzada y Ajustes

7.1. Parámetros ajustables en el SGBD

- Buffers, cachés, tamaño de transacciones.
- Configuración del planificador de consultas.

7.2. Escalado y alta disponibilidad

- SQL: Clustering, replicación master-slave o multi-master.
- NoSQL: Sharding, réplica de nodos, tolerancia a particiones.

7.3. Optimización en entornos cloud

- Autoescalado y provisión dinámica (ej. MongoDB Atlas, AWS RDS).
- Coste y rendimiento: uso racional de recursos.

8. Tendencias y Futuro

8.1. Inteligencia Artificial en bases de datos

- Motores de optimización automática de consultas con ML.
- Ejemplo: Self-driving databases (Oracle Autonomous DB).

8.2. Bases de datos multimodelo

- Integración de distintos modelos (documentos, grafos, relacional) en un solo motor (ej: ArangoDB, OrientDB).

8.3. Integración con ecosistemas Big Data

- Interacción con Spark, Hadoop, Kafka.
- Bases distribuidas como ClickHouse o Apache Drill.

9. Conclusiones

9.1. Recapitulación

- La manipulación eficiente de datos es crítica para el rendimiento global.
- SQL y NoSQL ofrecen soluciones complementarias.
- La elección depende de los requisitos funcionales y no funcionales.

9.2. Recomendaciones

- Analizar el patrón de acceso a datos antes de elegir SGBD.
- Medir, optimizar y volver a medir.
- Priorizar el mantenimiento y la escalabilidad futura.

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

1. Introducción

El lenguaje SQL (Structured Query Language) es la piedra angular en la gestión de bases de datos relacionales. Su relevancia se mantiene en entornos modernos como el Big Data, la computación en la nube o la inteligencia artificial, gracias a su capacidad de integrarse con lenguajes como Python, R o JavaScript.

Nuevas tecnologías han impulsado variantes y extensiones del SQL, así como procesos de transpilación a otros lenguajes (NoSQL, GraphQL) para adaptarse a distintos modelos de datos.

2. DDL – Lenguaje de definición de datos

Tipos de datos modernos

- JSON, XML: Permiten almacenar estructuras complejas y jerárquicas.
- GIS: Soporte geoespacial para mapas, coordenadas y rutas.
- Tipos personalizados: Definidos por el usuario, útiles para ML y estadísticas.

Tablas avanzadas

- Distribuidas y particionadas: Dividen datos entre servidores, favoreciendo la escalabilidad.
- Temporales: Uso limitado en el tiempo; útiles en procesamiento en tiempo real.
- En memoria: Almacenamiento en RAM para mayor velocidad.

Vistas avanzadas

- Materializadas: Precalculan resultados para mejorar el rendimiento.
- En tiempo real: Reflejan cambios instantáneamente.
- Seguras: Limitan el acceso según roles y permisos.

Restricciones modernas

- Triggers avanzados: Reaccionan automáticamente a cambios de datos.
- Consistencia eventual vs. fuerte: Equilibrio entre velocidad y precisión en entornos distribuidos.
- Cumplimiento normativo: Implementación de estándares como GDPR o HIPAA.

3. DML – Lenguaje de manipulación de datos

Manipulación avanzada

- Datos semi-estructurados: Uso de JSON y XML directamente desde SQL.
- Consultas de streaming: Datos en flujo continuo (redes sociales, IoT).
- Funciones analíticas (window functions): Permiten cálculos como promedios móviles sin agrupar.

Optimización de consultas

- Índices avanzados: Full-text, espaciales o sobre datos jerárquicos.
- Particionamiento y sharding: Distribuyen datos para aumentar rendimiento.

- Caching y precomputación: Aceleran consultas frecuentes

Integración con análisis de datos

- Plataformas como Apache Spark, Snowflake o BigQuery permiten usar SQL sobre grandes volúmenes.
- Compatibilidad con TensorFlow, PyTorch: Aplicaciones de SQL para modelos predictivos y aprendizaje automático.

4. DCL – Lenguaje de control de datos

Seguridad granular

- Row-level y column-level security: Acceso restringido por filas o columnas.
- Auditoría y monitoreo: Seguimiento de accesos y modificaciones en tiempo real.
- Encriptación y anonimización: Protegen datos sensibles en cumplimiento de normativas.

5. SQL EMBEBIDO – Aplicaciones complejas

Integración distribuida

- Cursores optimizados: Manejo eficiente de grandes volúmenes.
- Transacciones ACID vs. BASE:
 - ACID: Alta fiabilidad (banca, sanidad).
 - BASE: Mayor escalabilidad (redes sociales, apps globales).

Aplicaciones modernas

- Blockchain: SQL sobre estructuras inmutables.
- GraphQL y REST APIs: Traducen peticiones SQL a interfaces modernas.
- Serverless: SQL ejecutado en funciones como AWS Lambda sin gestionar servidores.

6. Alternativas a SQL

NoSQL y NewSQL

- MongoDB, Cassandra: Modelo documental o clave-valor.
- NewSQL: Mantiene sintaxis SQL con mejoras de escalabilidad (ej. CockroachDB).

GraphQL

- Lenguaje de consultas flexible que solicita solo los datos necesarios.
- Compatible con SQL mediante transpilación y wrappers.

Lenguajes específicos de dominio (DSL)

- Pandas, R: Análisis estadístico avanzado.
- Flink, Kafka Streams: Procesamiento en tiempo real.

7. Optimización y transpilación

Optimización automática

- Bases de datos que aprenden patrones de uso y ajustan sus índices y planes de ejecución con IA.

Transpilación entre modelos

- Conversión de SQL a GraphQL o a sentencias compatibles con NoSQL.
- Herramientas que permiten portabilidad entre plataformas heterogéneas.

8. Aplicaciones actuales

Business Intelligence

- Integración con herramientas como Power BI, Tableau, Qlik.

- Bases en la nube permiten consultas en tiempo real y dashboards dinámicos.

Learning Analytics

- SQL para evaluar progreso académico, detectar patrones y aplicar modelos predictivos.
- Uso en plataformas como Moodle o Canvas para adaptar contenidos al alumnado.

9. Perspectivas futuras

- SQL + IA: Uso en sistemas de recomendación y aprendizaje personalizado.
- Bases de datos autónomas: Se autogestionan, optimizan y aseguran sin intervención humana.
- Bases multimodelo: Unifican documentos, grafos, columnas y tablas relacionales.
- Lenguajes híbridos: Combinan paradigmas relacional, documental y gráfico en una sola sintaxis.

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

1. Introducción

Concepto de riesgo, impacto y vulnerabilidad.

La seguridad de los datos es un pilar fundamental para proteger la información frente a accesos no autorizados, manipulaciones indebidas o pérdidas accidentales. Abarca medidas técnicas, organizativas y legales orientadas a preservar la:

- Confidencialidad (prevención de accesos no autorizados)
- Integridad (evitar modificaciones no autorizadas)
- Disponibilidad (acceso continuo a los datos)
- Autenticidad (verificación de identidad)
- No repudio (imposibilidad de negar acciones realizadas)

La protección de datos debe abordarse tanto desde la seguridad activa (prevención, detección temprana) como desde la seguridad pasiva (respuesta, recuperación).

2. Servicios de Seguridad Aplicados a los Datos

2.1 Confidencialidad

- Cifrado en tránsito y en reposo: TLS/SSL, IPsec, AES, RSA.
- Clasificación y etiquetado de la información por sensibilidad.
- Modelos de control de acceso: RBAC, ABAC, Zero Trust, ACLs.

2.2 Integridad

- Hashing criptográfico: SHA-256, SHA-3, HMAC.
- Firmas digitales: Verifican integridad y autenticidad.
- FIM (File Integrity Monitoring): Wazuh, AIDE, Tripwire.
- Registros de transacciones con trazabilidad completa.
- Restricciones de integridad en bases de datos: claves primarias/foráneas, restricciones CHECK, UNIQUE, NOT NULL, DEFAULT.
- Assertions y constraints empresariales para reglas de negocio complejas.

2.3 Disponibilidad

- Backup 3-2-1: Tres copias, dos formatos, una externa.
- Alta disponibilidad (HA): Clústeres, replicación, failover.
- Planes BCP y DRP: Resiliencia ante incidentes.

2.4 Autenticidad y No Repudio

- Autenticación fuerte: MFA, biometría, certificados.
- Timestamping y registros firmados: RFC 3161.

3. Técnicas de Protección de Datos

3.1 Cifrado Avanzado

- Cifrado en la nube: CMK, BYOK, HSM.
- Cifrado a nivel de campo, objeto y fila en bases de datos (TDE, Always Encrypted).

3.2 Prevención de Pérdida de Datos (DLP)

- Soluciones DLP: Microsoft Purview, Forcepoint.
- Bloqueo de canales de fuga: USB, correo, SaaS.

3.3 Enmascaramado y Pseudonimización

- Enmascaramado dinámico de datos: Para entornos no productivos.
- Anonimización y pseudonimización: Cumplimiento de RGPD y LOPDGDD.

3.4 Gestión de Accesos y Privilegios

- Principio de mínimo privilegio y necesidad de saber.
- IAM centralizado: Azure AD, Okta, LDAP.
- Revisión periódica de accesos y segregación de funciones.

4. Sistemas de Protección de Datos

4.1 Copias de Seguridad y Recuperación

- Copias completas, incrementales, diferenciales.
- Snapshots y versionado: AWS S3, ZFS, Azure Blob.
- Pruebas periódicas de restauración.

4.2 Monitorización y Auditoría

- SIEMs: Wazuh, ELK, Splunk.
- Alertas en tiempo real y correlación de eventos.
- Auditoría forense de logs.

4.3 Integridad de Archivos y Sistemas

- Hashes automatizados y firmas para verificación de integridad.
- Control de cambios y monitoreo de archivos críticos.

4.4 Seguridad en Bases de Datos

- Fundamentos de Seguridad
- Propiedades ACID: Garantizan transacciones confiables:
 - Atomicidad: Todo o nada.
 - Consistencia: Mantiene reglas y restricciones.
 - Aislamiento: Las transacciones no interfieren entre sí.
 - Durabilidad: Los datos se conservan tras un fallo.
- Transacciones seguras: Uso de COMMIT, ROLLBACK, control de concurrencia (LOCK, MVCC).
 - Gestión mediante **BEGIN, COMMIT, ROLLBACK**.
 - Uso en operaciones críticas como transferencias, actualizaciones encadenadas o ajustes de inventario.
 - Control de errores para garantizar coherencia y revertir cambios en fallos.

SQL:

- Roles y privilegios personalizados: Principio del menor privilegio.
- Triggers (disparadores): Automatización de auditorías, validaciones, controles internos.
- Restricciones de integridad (PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE).

- Assertions: Definición de reglas de negocio complejas a nivel de esquema.
- **Niveles de aislamiento** (según ANSI SQL):
 - **READ UNCOMMITTED**: Máxima concurrencia, mínima seguridad.
 - **READ COMMITTED**: Previene lecturas sucias (por defecto en muchos SGBD).
 - **REPEATABLE READ**: Evita lecturas no repetibles.
 - **SERIALIZABLE**: Mayor aislamiento, evita todas las anomalías.
- **Control de concurrencia**:
 - **Bloqueos (Locks)**: **SELECT ... FOR UPDATE**, bloqueos a nivel de fila o tabla.
 - **MVCC (Multiversion Concurrency Control)**: Lecturas consistentes sin bloqueos (PostgreSQL, Oracle).
 - **Deadlocks**: Prevención mediante acceso ordenado y detección automática.
- **Gestión de roles y privilegios personalizados**: Acceso granular por funciones.
- **Cifrado**:
 - **TDE (Transparent Data Encryption)**.
 - **Always Encrypted** para columnas sensibles.
- **Validación de entradas y protección frente a inyecciones SQL**: Uso de ORM y consultas parametrizadas.
- **Restricciones de integridad**:
 - **PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK**.
 - **Assertions**: Validaciones complejas sobre múltiples tablas.
- **Triggers (disparadores)**:
 - Auditoría automática de operaciones.
 - Verificación de reglas de negocio en tiempo real.
- **Alta disponibilidad**:
 - SQL Server AlwaysOn, PostgreSQL + Patroni, MySQL Galera Cluster.
- Protección frente a inyecciones SQL: ORMs, validación de inputs, consultas parametrizadas.

NoSQL:

- Control de acceso por roles: MongoDB, Couchbase.
- Cifrado en tránsito y reposo: MongoDB, Cassandra.
- Auditoría: MongoDB Ops Manager, CloudTrail (DynamoDB).
- Validación de esquemas: En bases tipo JSON como MongoDB.

4.5 Seguridad en Sistemas de Archivos

- Cifrado de disco/sistemas: LUKS, BitLocker, eCryptfs.
- Permisos granulares: NTFS (Windows), ACLs en ext4 y ZFS.
- Snapshots y versionado: ZFS, Btrfs, LVM.

4.6 Clústeres y Replicación

- Alta disponibilidad activa/pasiva: Failover automático.

Replicación:

- Síncrona: Máxima consistencia.
- Asíncrona: Mejor rendimiento, pero posible pérdida parcial.

- Balanceo de carga: Nginx, HAProxy, Round Robin DNS.

5. Estándares y Legislación

5.1 Estándares Internacionales

- ISO/IEC 27001 / 27002: Gestión de seguridad de la
- ISO/IEC 27018: Protección de datos personales en la nube.
- NIST: SP 800-53 (controles), SP 800-171 (datos controlados), SP 800-207 (Zero Trust).

5.2 Legislación

RGPD y LOPDGDD: Base legal, consentimiento, derechos del usuario.

ENS: Marco normativo en la Administración Pública Española.

5.3 Evaluación de Riesgos y Cumplimiento

- Metodologías: MAGERIT, ISO 27005.
- Herramientas: PILAR (CCN-CERT).
- DPIA: Evaluaciones de impacto en privacidad (art. 35 RGPD).

6. Amenazas a la Seguridad de los Datos

- Ransomware y malware destructivo.
- Amenazas internas (insider threats).
- Errores de configuración en cloud.
- Inyecciones (SQL, NoSQL, XSS).
- Man-in-the-Middle (MITM).
- Shadow IT y servicios no autorizados.

7. Seguridad de los Datos en la Nube

7.1 Riesgos Específicos en Cloud

- Buckets públicos, credenciales expuestas.
- Pérdida de visibilidad sobre datos en SaaS.
- Uso malicioso de recursos: cryptojacking.

7.2 Controles de Seguridad en Cloud

- CSPM: Defender for Cloud, Prisma Cloud.
- IAM, MFA, federación.
- Cifrado controlado por el cliente.
- Monitorización: CloudTrail, GuardDuty, Azure Monitor.

8. Formación, Buenas Prácticas y Concienciación

8.1 Políticas Internas

- Clasificación de datos.
- Plan de backup y recuperación.
- Gestión de incidentes.

8.2 Concienciación y Capacitación

- Formación continua en ciberseguridad.
- Simulacros de phishing y fuga de datos.
- Programas de INCIBE: CyberCamp, recursos educativos.

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

1. Introducción: Interfaces más allá de las pantallas.

- Definición: Punto de contacto entre personas y sistemas digitales.
- HCI / IMH: Diseño centrado en la persona.
- Tipos modernos de interfaz:
 - Gráficas (GUI): Web, móviles, escritorio.
 - Conversacionales: Chatbots, asistentes (Alexa, Siri).
 - Gestuales y sin contacto: Kinect, Leap Motion.
 - AR/VR/MR: Realidad Aumentada, Virtual y Mixta en educación, industria, ocio.
 - BCI: Interfaces cerebro-computadora (en investigación).

2. Usabilidad y experiencia de usuario (UX/UI)

- Usabilidad ≠ UX:
 - Usabilidad: Uso eficiente, efectivo y satisfactorio (heurísticas de Nielsen).
 - UX: Emociones y percepción global de la interacción.
- Métricas clave:
 - Efectividad: Tareas completadas correctamente.
 - Eficiencia: Tiempo/esfuerzo.
 - Satisfacción: Escalas como SUS (System Usability Scale).
- Accesibilidad (WCAG 2.2):
 - Contrastes, navegación por teclado, lectores de pantalla.
 - Normativa europea: obligatorio en servicios digitales públicos.
- Dark Patterns:
 - Diseños que manipulan (ej. botones engañosos, ocultar cancelación).

3. Principios y heurísticas de diseño

- Heurísticas de Nielsen:
 - Visibilidad del sistema, control, error prevention, ayuda contextual, etc.
- Leyes del diseño UX:
 - Fitts: Cuanto más grande y cerca, más rápido el clic.
 - Hick-Hyman: Muchas opciones = más tiempo de decisión.
- Gestalt y percepción visual:
 - Proximidad, semejanza, continuidad → coherencia visual.

4. Diseño visual moderno

- Psicología del color: Emociones, cultura, significado.
- Tipografía y jerarquía: Legibilidad, tamaños, peso visual.
- Motion UI y microinteracciones
 - Animaciones suaves para guiar, informar y deleitar.

5. Tipo de interfaces gráficas de usuario (GUI)

5.1 Web

- Diseño responsive: Grid, Flexbox, Mobile First.
- Componentes modernos:

- Material Design (Google), Fluent (Microsoft), Bootstrap, Tailwind CSS.

5.2 Móvil

- Guías oficiales:
 - Android: Material.
 - iOS: Human Interface Guidelines.
- Frameworks populares: React Native, Flutter, SwiftUI.
- PWA: Apps web que funcionan como nativas.

5.3 Escritorio

- SDI / MDI: Interfaz de documento único o múltiple.
- Frameworks: Electron, WPF, Qt.

5.4 Emergentes

- Wearables: Smartwatches, gafas inteligentes.
- Smart TVs: Interfaces simplificadas para mando o voz.

6. Proceso de diseño UX/UI

6.1 Metodologías

- Design Thinking: Empatía, ideación, prototipado.
- Lean UX: Ciclos rápidos, feedback constante.

6.2 Técnicas

- Personas: Perfiles representativos de usuarios.
- User Journeys: Flujo completo de experiencia.
- Wireframes y Mockups:
 - Herramientas: Figma, Adobe XD, Sketch.
- Prototipado interactivo:
 - Test con usuarios reales, iteraciones, A/B Testing.

7. Desarrollo de interfaces modernas

Web

- Tecnologías base:
 - HTML5, CSS3, JavaScript ES6+.
 - Diseño responsive: CSS Grid, Flexbox, Custom Properties.
- Frameworks web modernos:
 - React, Vue, Svelte, Angular.
 - Componentes reutilizables y declarativos.
- Gestión de estado:
 - Redux, Zustand, Recoil, Vuex.
- Single Page Applications (SPA) y Progressive Web Apps (PWA).

Aplicaciones Móviles

- Frameworks multiplataforma:
 - React Native, Flutter, Ionic: código único para Android e iOS.
- Desarrollo nativo:
 - Android: Java, Kotlin + Jetpack Compose.
 - iOS: Swift + SwiftUI.
- PWA: apps web con experiencia nativa, sin necesidad de instalar.

Aplicaciones de Escritorio

- Frameworks multiplataforma:
 - Electron.js: JS + HTML + CSS para apps nativas (ej. VS Code, Slack).
 - Tauri: alternativa más ligera a Electron.
- Frameworks nativos:
 - WPF / WinUI (Windows), GTK+ / Qt (Linux), AppKit / SwiftUI (macOS).
- UI declarativa y reactiva: tendencia común (Compose, SwiftUI, React, etc.).

Interfaces para Dispositivos Emergentes

- Smart TVs:
 - Desarrollo con Tizen (Samsung), WebOS (LG), o HTML5.
- Wearables:
 - watchOS (Apple), Wear OS (Google).
 - Interfaz reducida, orientada a gestos y voz.
- Realidad Aumentada y Virtual:
 - Unity + C#, Unreal + Blueprints, ARKit (iOS), ARCore (Android).
- Interfaces por voz / conversación:
 - Alexa Skills, Google Actions, chatbots con NLP e IA generativa.

8. Tendencias futuras

- Interacción multimodal: Voz, gestos, pantallas táctiles.
- Interfaces generadas por IA:
 - Diseño automático con Framer AI, Galileo AI.
- Realidad Aumentada, Mixta y Hologramas.
- UX personalizado con Big Data:
 - Análisis de comportamiento con ML.
- Automatización del diseño UX:
 - Algoritmos que ajustan interfaces en tiempo real según el usuario

Tema 60: Sistemas basados en el conocimiento. Representación del conocimiento. Componentes y arquitectura.

1. Introducción a los Sistemas Cognitivos Inteligentes

1.1. De los sistemas expertos a la inteligencia híbrida

- Primera generación: Sistemas expertos (ej: MYCIN, DENDRAL) con reglas simbólicas explícitas.
- Segunda generación: Sistemas neuro-simbólicos que combinan reglas lógicas + modelos subsimbólicos (ML/DL).
- Ejemplos clave:
 - IBM Watson Health: razonamiento clínico + procesamiento de lenguaje natural (NLP).
 - AlphaCode y AlphaFold 2 (DeepMind): redes neuronales guiadas por principios simbólicos.

1.2. Machine Learning como motor cognitivo

- Aprendizaje automático (ML) permite el descubrimiento de patrones, generalización y adaptación.
- Deep Learning para señales complejas (imágenes, audio, texto).
- Modelos explicables (XAI) para integración con entornos sensibles (sanidad, derecho, finanzas).

2. Representación del Conocimiento en la Era de la IA

2.1. Métodos simbólicos y neuro-simbólicos

Método	Propósito	Ejemplo moderno
Ontologías	Formalizar conceptos y relaciones	SNOMED CT, OWL (Web Ontology Language)
Grafos de conocimiento	Mapear relaciones complejas	Google Knowledge Graph, Amazon Neptune
Reglas lógicas	Inferencia explicable	Pyke, CLIPS, Prolog
Embeddings semánticos	Convertir conocimiento en vectores	BERT + KG embeddings, RDF2Vec

2.2. Representación híbrida

- TensorFlow Knowledge Graphs: embeddings estructurados.
- DeepProbLog: lógica probabilística combinada con redes neuronales.
- Markov Logic Networks (MLN): modelos probabilísticos + lógicos.

3. Arquitectura de Sistemas Basados en Conocimiento (SBC)

3.1. Componentes clásicos + extensiones modernas

graph TD

A[Usuario] --> B[Captura del conocimiento]

B --> C[Base de conocimiento]

C --> D[Motor de inferencia]

D --> E[Decisión lógica]

E --> F[Modelo ML]

F --> G[Resultado enriquecido]

G --> H[Explicación (XAI)]

- Base de conocimiento (KB): reglas, hechos, ontologías, grafos.
- Motor de inferencia: aplica reglas, ejecuta lógica de primer orden.
- Modelo ML: clasifica, predice o filtra información no estructurada.
- Sistema explicativo: devuelve resultado + justificación (SHAP, LIME, Trepan, XAI).

3.2. Flujo híbrido típico

1. Captura del conocimiento (experto o automático).
2. Aplicación de reglas explícitas (símbolos).
3. Refinamiento con modelo ML.
4. Fusión de resultados y explicación al usuario.

4. Machine Learning aplicado a SBC

4.1. Enriquecimiento automático de conocimiento

- AutoML + Clustering: generación de reglas desde datos (ej: diagnóstico de fallos en fábricas).
- Fine-tuning de LLMs (como GPT, Claude): para creación de ontologías personalizadas.
- Named Entity Recognition (NER) + Graph Construction desde textos científicos.

4.2. Motores de inferencia mejorados

- Neuro-simbólicos:
 - AlphaGeometry (DeepMind): geometría matemática simbólica + redes neuronales.
 - NeuroSymbolic Concept Learner (MIT): visión + lógica para razonamiento sobre objetos.
- Probabilistic programming: Pyro, Stan, DeepProbLog.

4.3. Interfaces inteligentes y NLP

- Captura de conocimiento mediante procesamiento de lenguaje natural (ChatGPT, spaCy, HuggingFace).
- Transformación de texto libre en conocimiento estructurado.
- Interfaces con visualización dinámica de razonamiento (Graphviz, D3.js, Lucidchart).

5. Aplicaciones Modernas con SBC + ML

5.1. Salud y diagnóstico inteligente

- Caso real: Mayo Clinic → Sistema híbrido con:
 - 12.000 reglas clínicas (guías médicas)
 - CNNs para imagen médica
 - NLP para interpretación de notas clínicas
 - Reducción del error diagnóstico en un 37%.

5.2. Banca y seguros

- Pipeline típico:
 1. Reglas regulatorias (AML, GDPR)
 2. Modelos ML para scoring de crédito (XGBoost, LightGBM)
 3. Generación de explicaciones para cumplimiento normativo (SHAP, LIME)

5.3. Industria 4.0 y mantenimiento predictivo

- Digital Twins con grafos de conocimiento + LSTM.
- Predicción de fallos con ML y reglas temporales.
- Casos: Siemens, Bosch, Airbus.

6. Tendencias Emergentes

6.1. Generative AI + Bases de conocimiento

- LLMs adaptados con dominios específicos:
 - Ej: GPT con vector stores semánticos (Pinecone, Weaviate)
 - Uso de retrieval-augmented generation (RAG) para respuestas precisas.

6.2. Aprendizaje continuo autónomo

- Detectores de data drift y concept drift.
- Sistemas que reajustan reglas automáticamente (AutoML + Reasoners).

6.3. Quantum AI para conocimiento simbólico

- QNN (Quantum Neural Networks) aplicados a grafos de conocimiento complejos.
- Simulación de inferencia lógica en arquitecturas cuánticas (investigación en IBM Q, Xanadu AI).

7. Ejemplo Práctico: Integración SBC + ML en Python

```
from pyke import knowledge_engine
from sklearn.ensemble import RandomForestClassifier
```

```
# 1. Motor de reglas
```

```
engine = knowledge_engine.engine(__file__)
engine.activate('diagnostico')
```

```
# 2. Modelo ML entrenado
```

```
model = RandomForestClassifier()
model.fit(X_train, y_train)
```

```
# 3. Fusión
```

```
def diagnosticar(paciente):
    reglas = engine.prove_1('diagnostico', paciente)
    pred_ml = model.predict([paciente['datos']])
    return combinar(reglas, pred_ml)
```

Extensiones modernas:

- Integración con LangChain, OpenAI API, HuggingFace.
- Visualización de resultados con Gradio, Streamlit.

8. Evaluación Comparativa y Métricas de Éxito

Métrica	SBC clásico	SBC+ML avanzado
---------	-------------	-----------------

Precisión diagnóstica	70%	> 90%
Adaptabilidad	Baja	Alta (aprendizaje continuo)
Coste de mantenimiento	Alto	Medio (AutoML)
Explicabilidad	Muy alta	Alta (XAI integrado)
Escalabilidad	Limitada	Alta (cloud-native)

9. Conclusiones Finales

- Sistemas Basados en Conocimiento siguen siendo cruciales en sectores críticos (salud, derecho, industria).
- La fusión con IA moderna permite superar el dilema entre rendimiento y explicabilidad.
- En 2025, las arquitecturas neuro-simbólicas son la vía dominante en entornos donde los errores tienen un alto coste.
- La ingeniería del conocimiento, complementada con habilidades en IA, es una de las profesiones más demandadas.

Tema 61: Redes y Servicios de Comunicaciones

1. Introducción

1.1. Importancia de las redes en la sociedad digital

- Infraestructura esencial para servicios: salud, industria, comercio, gobierno.
- Soporte de tecnologías disruptivas: IoT, IA, cloud, big data, 5G.
- Impacto directo en la economía, ciberseguridad, educación y sostenibilidad.

1.2. Objetivos y alcance del tema

- Analizar los fundamentos, evolución y arquitectura de las redes.
- Comprender los servicios de comunicación y sus parámetros de calidad.
- Evaluar aspectos de seguridad, normativos y tendencias tecnológicas.

1.3. Metodología de estudio

- Enfoque técnico-práctico, orientado a estándares (ISO, ITU, NIST).
- Relación entre arquitectura, servicios y ciberseguridad.
- Enlace con legislación TIC y transformaciones digitales actuales.

2. Redes de Comunicaciones

2.1. Sistemas de comunicación

2.1.1. Elementos básicos

- Emisor, receptor, mensaje, canal de transmisión, protocolo, ruido.

2.1.2. Parámetros de calidad

- Ancho de banda: Capacidad máxima de transmisión (Mbps/Gbps).
- Latencia: Tiempo de ida y vuelta de un paquete (ms).
- Jitter: Variación en la latencia, afecta a tráfico en tiempo real.
- Pérdida de paquetes: Impacto en calidad de voz/vídeo y fiabilidad.
- Tasa de error de bits (BER): Precisión de la transmisión (más crítica en entornos físicos como fibra o radio).
- QoS (Calidad de Servicio): Conjunto de mecanismos para garantizar rendimiento. Ej: priorización de VoIP frente a datos no críticos.

2.2. Evolución histórica

2.2.1. De los orígenes a ARPANET

- Telégrafo, telefonía, conmutación de circuitos → ARPANET → Internet.

2.2.2. Generaciones de redes móviles (1G-6G)

- 1G (analógica), 2G (SMS, voz digital), 3G (datos), 4G (banda ancha móvil), 5G (latencia ultra baja, IoT masivo), 6G (en desarrollo).

2.2.3. Futuro: IoT, redes cuánticas, edge computing, redes auto-gestionadas (SDN, IA).

2.3. Tipos de redes por extensión

2.3.1. PAN: Bluetooth, Zigbee, NFC.

2.3.2. LAN: Ethernet, WiFi 6/6E, redes cableadas de ámbito local.

2.3.3. MAN: Interconexión de redes LAN a escala urbana.

2.3.4. WAN: Redes de alcance mundial (Internet, MPLS, SD-WAN).

2.4. Tipos de redes por ámbito

2.4.1. Públicas vs. privadas: Control de acceso, seguridad, disponibilidad.

2.4.2. VPNs: Tecnologías como IPsec (nivel 3) y SSL/TLS (nivel 7) para asegurar conexiones remotas.

2.5. Topologías de red

2.5.1. Estrella, bus, anillo, malla, híbridas.

2.5.2. Comparativa: fiabilidad, coste, facilidad de mantenimiento, escalabilidad.

2.6. Tecnologías de red

2.6.1. WiFi (802.11ac/ax), cifrado WPA3, seguridad en redes inalámbricas.

2.6.2. Redes móviles: LTE, VoLTE, 5G (mmWave, NSA/SA).

2.6.3. Fibra óptica: FTTH, GPON, DWDM (para redes troncales de alta capacidad).

2.7. Componentes de red

2.7.1. Hardware: Routers, switches, firewalls, APs, balanceadores.

2.7.2. Software: Sistemas de monitorización (Zabbix, Nagios), gestión (SNMP, NetFlow), automatización (Ansible, SDN).

2.8. Modelos y protocolos

2.8.1. Modelo OSI vs. TCP/IP

- Capas, funciones, interoperabilidad.

2.8.2. Protocolos clave:

- TCP, UDP, IP, HTTP/HTTPS, DNS, DHCP, ICMP, ARP.

3. Medios de Transmisión y Redes Móviles

3.1. Medios de transmisión

3.1.1. Cableado: UTP (cat5e, cat6), coaxial (DOCSIS).

3.1.2. Inalámbrico: Radiofrecuencia, satélite (GEO, LEO).

3.1.3. Fibra óptica:

- Monomodo (larga distancia), multimodo (centros de datos).
- Aplicaciones: FTTH, backbone de red, interconexión de datacenters.

3.2. Redes móviles

3.2.1. 2G/2.5G: GSM, GPRS, EDGE.

3.2.2. 3G/3G+: UMTS, HSPA.

3.2.3. 4G/4G+: LTE, LTE-A, VoLTE.

3.2.4. 5G:

- Latencia <1 ms, IoT masivo, conectividad ultra densa.
- Arquitectura: Core 5G, slices, edge computing.

4. Servicios de Comunicaciones

4.1. Correo electrónico

- Protocolos: SMTP, POP3, IMAP.
- Seguridad: SPF, DKIM, DMARC, cifrado S/MIME y PGP.

4.2. Navegación web

- HTTP, HTTPS (TLS 1.3), certificados digitales.
- DNS, DNSSEC, DoH (DNS over HTTPS), CDNs (Akamai, Cloudflare).

4.3. Transferencia de archivos

- FTP, SFTP, SCP, P2P.
- Servicios cloud: Dropbox, Google Drive, OneDrive, servicios de backup.

4.4. Voz y datos

- VoIP: SIP, RTP, codecs.
- Mensajería instantánea (Signal, WhatsApp, XMPP, Matrix).

4.5. Otros servicios

- IoT: Protocolos MQTT, CoAP.
- Cloud computing: SaaS, PaaS, IaaS; ejemplos: AWS, Azure, GCP.

5. Seguridad en Redes

5.1. Amenazas

- Malware, DDoS, MITM, spoofing, sniffing, ransomware.
- Específicas de redes móviles, IoT y entornos industriales.

5.2. Protección

- Firewalls, IDS/IPS, segmentación, cifrado (TLS, IPsec, WPA3).
- Zero Trust, microsegmentación, gestión de accesos (NAC, 802.1X).

5.3. Normativas y estándares

- GDPR, LOPDGDD, ENS.
- Buenas prácticas: NIST SP 800-53, ISO/IEC 27001 y 27033.

6. Legislación

6.1. Ley General de Telecomunicaciones

- Regulación de operadores, uso del espectro, interoperabilidad.

6.2. Neutralidad de la red

- No discriminación del tráfico, marco normativo europeo.

7. Conclusiones

7.1. Resumen de conceptos clave

- Redes como base de la economía digital.
- Necesidad de garantizar calidad, seguridad, disponibilidad y cumplimiento.

7.2. Tendencias futuras

- 6G, redes cuánticas, SDN, redes autónomas, smart cities.
- Ciberresiliencia y automatización como ejes del futuro digital.

Tema 72: Seguridad en Sistemas en Red: Servicios, Protecciones y Estándares Avanzados

1. Introducción

La seguridad en sistemas en red es fundamental para garantizar la protección de los activos de información frente a amenazas internas y externas. Se basa en asegurar la confidencialidad, integridad y disponibilidad (CID) de los sistemas y datos a través de estrategias de prevención, detección y respuesta.

2. Servicios de Seguridad

Los servicios de seguridad proporcionan los mecanismos fundamentales para proteger la información y los sistemas:

2.1 Autenticación y Autorización

- Métodos: Contraseñas, certificados, OTP, biometría, tokens.
- SSO y MFA: Single Sign-On (Keycloak, Okta) y autenticación multifactor (Google Authenticator, YubiKey).
- Protocolos: LDAP, RADIUS, Kerberos, SAML, OAuth2, OpenID Connect.

2.2 Control de Acceso

- Modelos: DAC, MAC, RBAC, ABAC.
- Control del acceso al medio: Filtrado MAC, IP, VLANs, portales cautivos, NAC (Cisco ISE, FortiNAC).

2.3 Cifrado y No Repudio

- Cifrado en tránsito: TLS/SSL, IPsec, HTTPS, SSH, VPNs (OpenVPN, WireGuard).
- Cifrado en reposo: AES-256, cifrado de discos y bases de datos.
- Firmas digitales y certificados X.509.

2.4 Auditoría y Registro

- SIEM: Splunk, ELK Stack, Wazuh.
- Recolección y correlación de logs.
- Análisis de comportamiento y detección de anomalías.

3. Técnicas de Protección

Estas técnicas permiten reducir la superficie de ataque y controlar el riesgo de amenazas conocidas y emergentes:

3.1 Segmentación

- Lógica: VLANs por tipo de usuario/servicio.
- Física: Redes separadas por funciones críticas.
- Microsegmentación: Políticas específicas por VM/contenedor.
- SDN: Centralización y automatización de reglas.

3.2 Bastionado (Hardening)

- Eliminación de servicios innecesarios.
- Configuración segura de sistemas y redes.
- Herramientas: CIS Benchmarks, OpenSCAP, Ansible, Lynis.
- Integración con Splunk/Wazuh para monitorización continua.

3.3 Prevención de Amenazas

- Antivirus/EDR: CrowdStrike, SentinelOne.
- MFA: Bloqueo de accesos no autorizados.
- DNSSEC, DoH/DoT: Prevención de spoofing.

- Bloqueo por listas negras y reputación IP.

4. Sistemas de Protección

Infraestructura y soluciones especializadas para prevenir, detectar y responder a ataques:

4.1 Cortafuegos (Firewalls)

- Capa 3/4: iptables, nftables, Cisco ASA.
- Capa 7: WAF (ModSecurity, AWS WAF), proxies (Squid).
- Next-Gen Firewalls: Fortinet, Palo Alto, con inspección profunda, control de apps, IPS.

4.2 IDS / IPS

- IDS: Snort, Suricata, Zeek. Detectan actividades anómalas o maliciosas.
- IPS: Actúa en tiempo real para bloquear ataques (Suricata inline, Cisco Firepower).
- FIM (File Integrity Monitoring): AIDE, Tripwire, Wazuh.

4.3 Sistemas de Backup y Recuperación

- Estrategia 3-2-1.
- Herramientas: Veeam, Bacula, Rsync, Borg.
- Planes de continuidad (BCP) y recuperación ante desastres (DRP).

5. Estándares, Normativas y Legislación

5.1 Estándares Internacionales

- ISO/IEC 27001: SGSI.
- NIST SP 800-53 / CSF / 800-171: Controles y marcos para organizaciones de EE.UU.
- COBIT 2019: Gobierno de TI.
- MITRE ATT&CK: Tácticas y técnicas adversarias.

5.2 Legislación y Regulaciones

- RGPD / LOPDGDD: Protección de datos personales.
- ENS: Esquema Nacional de Seguridad (España).
- NIS2: Requisitos para sectores esenciales.
- PCI DSS, HIPAA, SOX (según el sector).

5.3 Evaluación de Riesgos y Cumplimiento

- MAGERIT: Metodología oficial.
- PILAR: Herramienta del CCN-CERT para análisis de riesgos, amenazas e impacto.

6. Amenazas Comunes y Emergentes

- Hombre en el Medio (MITM): Cifrado, VPN, HSTS como defensa.
- Cloud Attacks: Exposición de recursos, abuso de facturación, criptojacking.
- Ransomware: Aislamiento, backups, EDR.
- DDoS/DoS: Mitigación con balanceadores, servicios anti-DDoS (Cloudflare, AWS Shield).

7. Formación y Concienciación

7.1 Actividades Educativas

- Simulaciones de ataques (phishing, ransomware).
- Talleres de concienciación.
- Políticas de uso seguro de la red.

7.2 Iniciativas de INCIBE (España)

- CyberCamp, HackOn, Cibercooperantes.
- Retos CTF, competiciones estudiantiles.
- Evaluación con PILAR y formación en metodologías de análisis de riesgo.

Tema 74: Sistemas Multimedia

1. Introducción a los Sistemas Multimedia

1.1. ¿Qué es un sistema multimedia?

- Definición: sistemas que integran múltiples tipos de medios digitales (texto, imagen, audio, vídeo, animación) en un entorno interactivo.
- Componentes esenciales: adquisición, codificación, almacenamiento, transmisión, decodificación y presentación.

1.2. Relevancia actual y contexto global

- El 82% del tráfico global en Internet es vídeo (Cisco, 2023).
- Aplicaciones: educación, entretenimiento, salud, transporte, smart cities.
- Nuevas fronteras: sistemas multimodales y multisensoriales en entornos XR, metaversos y entornos IoT.

1.3. Breve evolución histórica

- Etapas clave:
 - CD-ROM interactivo (90s)
 - Streaming adaptativo (2000s)
 - Realidad aumentada y sistemas XR (2020s)
- Hitos tecnológicos:
 - MP3 (1993), MPEG-4 (1998), YouTube (2005), AV1 (2018), Vision Pro (2023)

2. Fundamentos Técnicos de Representación y Procesamiento

2.1. Representación digital de medios

- Imagen:
 - Raster (JPEG, PNG, BMP) vs vectorial (SVG).
 - Profundidad de color, espacios (RGB, YUV), canal alfa.
- Audio:
 - Tasa de muestreo (Nyquist), cuantización, mono/estéreo, bitrate.
 - Formatos: WAV (sin compresión), FLAC (lossless), AAC (lossy).
- Vídeo:
 - FPS, resolución (4K, 8K), GOP (Group of Pictures), HDR.
 - Contenedores (MP4, MKV), códecs (H.264, H.265, AV1, VVC).

2.2. Fundamentos matemáticos aplicados

- Transformadas: Fourier (espectro de frecuencias), DCT (MPEG), Wavelet (JPEG2000).
- Álgebra lineal:
 - Convoluciones, kernels, filtros espaciales, operaciones morfológicas.
 - Uso en visión por computador y redes neuronales convolucionales (CNNs).

3. Tecnologías Clave en Codificación y Transmisión

3.1. Compresión y códecs modernos

Códec	Compresión	Aplicación	Soporte hardware
JPEG	10:1	Fotografía	Total
H.264	30:1	Streaming HD	Universal

H.265	50% mejor que H.264	4K/8K	Alta gama
AV1	30-40% < H.265	Web, Edge	NVIDIA, Intel
VVC (H.266)	50% < H.265	8K UHD	En expansión
LCEVC	Mejora dinámica	OTT, gaming	Versátil

3.2. Streaming adaptativo y transmisión

- Protocolos:
 - HLS (Apple), DASH (MPEG) – adaptan resolución al ancho de banda.
 - RTMP/RTSP para baja latencia.
- Técnicas: buffering predictivo, ABR (Adaptive Bitrate), CDN inteligente (Cloudflare, Akamai).

4. Arquitecturas y Procesamiento de Sistemas Multimedia

4.1. Flujo funcional típico

graph TD

```

A[Adquisición de medios] --> B[Codificación]
B --> C[Almacenamiento y empaquetado]
C --> D[Transmisión en red]
D --> E[Decodificación y presentación]

```

4.2. Ejemplos reales de arquitectura

- Zoom / Teams:
 - Codificación en tiempo real con Opus y VP9
 - Redirección SFU (Selective Forwarding Unit)
 - Cancelación de ruido con RNNs embebidas
- TikTok:
 - Ingesta multimedia + clasificación ML
 - Algoritmos de recomendación basados en señales audiovisuales

5. Integración con Inteligencia Artificial

5.1. IA Generativa y modelos multimodales

- Imagen:
 - Stable Diffusion, DALL·E 3: texto → imagen
 - NeRF + GANs: escenas 3D fotorrealistas
- Audio:
 - AudioLM, MusicLM (Google): generación semántica de audio
 - Clonación de voz (ElevenLabs, Meta Voicebox)

5.2. Análisis inteligente multimedia

- Visión por Computador:
 - Detección de objetos con YOLOv8, DETR.
 - Clasificación de escenas, estimación de profundidad (Monodepth).
- NLP multimodal:
 - CLIP, BLIP, GPT-4V: comprensión cruzada entre texto, imagen y vídeo.
 - Visual Question Answering, búsqueda semántica visual.

6. Desarrollo de Aplicaciones Multimedia

6.1. Herramientas y frameworks profesionales

- FFmpeg: manipulación de streams, edición avanzada CLI.
- OpenCV: análisis de imágenes, detección facial, filtros.
- WebRTC: transmisión en tiempo real peer-to-peer.
- MediaPipe (Google): pose estimation, tracking facial en dispositivos móviles.

6.2. Aceleración y optimización

- NVIDIA Video Codec SDK, Intel Quick Sync: transcodificación por hardware.
- GPU inference para modelos de IA en tiempo real.
- TensorRT / ONNX Runtime: inferencia rápida en Edge.

7. Tendencias Futuras y Desafíos Éticos

7.1. Multimedia inmersiva y XR

- Dispositivos XR: Meta Quest 3, Apple Vision Pro.
- Tecnologías emergentes:
 - Códecs volumétricos (MIV, V-PCC)
 - Codificación neural (Neural Radiance Fields)

7.2. Multimedia en Edge + 5G

- Codificación y análisis on-device.
- Ej: vigilancia inteligente, streaming AR en móviles con latencias <10ms.
- Plataformas: NVIDIA Jetson, Google Coral, AWS Wavelength.

7.3. Ética, privacidad y sesgos

- Deepfakes: detección con redes siamesas, blockchain para trazabilidad.
- Sesgos en reconocimiento facial (ej: datasets no representativos).
- Legislación: IA Act (UE), regulaciones sobre manipulación audiovisual.

8. Casos de Estudio Reales

8.1. Videovigilancia inteligente

- Adquisición: cámaras 4K HDR con visión nocturna.
- Procesamiento: YOLOv8 para personas, Whisper para audio.
- Almacenamiento: códec AV1, discos NVMe cifrados.
- Búsqueda semántica: CLIP + lenguaje natural ("hombre con chaqueta azul").
- Interfaz: visualización web + notificaciones móviles.

8.2. Producción de contenido para plataformas

- IA generativa para fondos de vídeo y subtítulos automáticos.
- Mejora de calidad con super-resolution (Real-ESRGAN).
- Clasificación automática para recomendaciones (embeddings multimodales).

9. Demostración Técnica Avanzada

9.1. Streaming adaptativo con Python y PyAV

```
import av, dash
def adapt_stream(source):
    container = av.open(source)
    for packet in container.demux(video=0):
        frame = packet.decode_one()
        if frame.pts % 10 == 0:
            adjust_quality()
        yield frame
```

9.2. Script para detección de deepfakes

```
from deepface import DeepFace
```

```
result = DeepFace.analyze("video_frame.jpg", actions=["emotion", "deepfake"])
```

10. Conclusiones Estratégicas

- Los sistemas multimedia modernos no son solo reproductores, sino plataformas cognitivas capaces de percibir, razonar y actuar.
- La convergencia con IA generativa, edge computing y XR marcará la próxima década.
- Las competencias clave:
 - Dominio de codecs + IA + análisis multimodal.
 - Enfoque ético, explicabilidad y sostenibilidad tecnológica

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.