

Oposiciones cuerpo de secundaria.

# **Esquemas dos páginas sobre temario oposición profesorado Secundaria.**

## **Especialidad informática**



Autor: Sergi García Barea

Actualizado Mayo 2025

## Licencia



### **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

<b>Introducción</b>	<b>2</b>
<b>Para el buen docente</b>	<b>2</b>
<b>¿Para qué prueba están adaptados estos esquemas?</b>	<b>2</b>
<b>Tema 1: Representación y comunicación de la información</b>	<b>3</b>
<b>Tema 2: Elementos funcionales de un ordenador digital. Arquitectura.</b>	<b>5</b>
<b>Tema 3: Componentes, estructura y funcionamiento de la Unidad Central de Proceso (CPU)</b>	<b>7</b>
<b>Tema 4: Memoria interna: Tipos, Direccionamiento, Características y funciones</b>	<b>9</b>
<b>Tema 10: Representación interna de los datos</b>	<b>11</b>
<b>Tema 11: Organización lógica de los datos. Estructuras estáticas.</b>	<b>13</b>
<b>Tema 12: Organización lógica de los datos. Estructura dinámicas.</b>	<b>15</b>
<b>Tema 13: Ficheros. Tipos, Características, organizaciones</b>	<b>17</b>
<b>Tema 16: Sistemas operativos: Gestión de procesos</b>	<b>19</b>
<b>Tema 20: Explotación y administración de sistemas operativos monousuario y multiusuario</b>	<b>21</b>
<b>Tema 21: Sistemas informáticos: Estructura física y funcional</b>	<b>23</b>
<b>TEMA 22: PLANIFICACIÓN Y EXPLOTACIÓN DE SISTEMAS INFORMÁTICOS</b>	<b>25</b>
<b>1. PARTE CIENTÍFICA</b>	<b>25</b>
<b>TEMA 23: DISEÑO DE ALGORITMOS. TÉCNICAS DESCRIPTIVAS</b>	<b>27</b>
<b>TEMA 24: LENGUAJES DE PROGRAMACIÓN: TIPOS Y CARACTERÍSTICAS</b>	<b>29</b>
<b>TEMA 25: PROGRAMACIÓN ESTRUCTURADA. ESTRUCTURAS BÁSICAS, FUNCIONES Y PROCEDIMIENTOS</b>	<b>31</b>
<b>TEMA 26: PROGRAMACIÓN MODULAR. DISEÑO DE FUNCIONES, RECURSIVIDAD Y LIBRERÍAS</b>	<b>33</b>
<b>Tema 27: Programación orientada a objetos</b>	<b>35</b>
<b>Tema 31: Lenguaje C. Estructura, Funciones, Entorno y Depuración</b>	<b>37</b>
<b>Tema 32: Manipulación de estructuras en C. Punteros, funciones</b>	<b>39</b>
<b>Tema 34: Sistemas Gestores de Bases de Datos. Funciones, componentes, arquitecturas y tipos</b>	<b>41</b>
<b>Tema 35: La definición de datos. Niveles, lenguajes y diccionario</b>	<b>43</b>
<b>Tema 36: La manipulación de datos: Modelado, Consultas y Optimización</b>	<b>45</b>
<b>Tema 38: Modelo de datos relacional. Estructuras, operaciones y álgebra relacional</b>	<b>47</b>
<b>Tema 72: Seguridad en sistemas de red: Servicios, protecciones, estándares avanzados</b>	<b>49</b>
<b>Tema 74: Sistemas multimedia</b>	<b>51</b>

## **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**

### **1.1 Introducción**

- Definición: transformación de fenómenos del mundo real en estructuras digitales binarias.
- Fundamento para todo procesamiento informático: desde el código hasta el hardware.

### **1.2 Sistemas de Numeración**

Base, dígitos, sistema posicional.

Sistemas: binario, octal, hexadecimal y decimal.

Conversión entre sistemas para debugging, direccionamiento y arquitectura.

### **1.3 Representación de Datos**

- **Enteros:**
  - Códigos: signo+magnitud, CA1 y CA2 (uso de CA2 por su simplicidad en hardware).
- **Punto flotante (IEEE 754):**
  - Precisión simple (32 bits) y doble (64 bits), compuestas por signo (1 bit), exponente (8/11 bits) y mantisa (23/52 bits).
  - Se normaliza desplazando el punto binario para que quede en la forma 1.xxxxx, permitiendo maximizar la precisión y garantizar una representación única.
- **Texto:**
  - ASCII (7 bits, limitado), 8 bits extendido y Unicode (UTF-8/16/32) para múltiples idiomas y emoji.
- **Imágenes:**
  - Representación como matriz de píxeles con RGB + canal alfa.
  - Formatos: BMP, PNG (sin pérdida), JPEG (con pérdida).
- **Vídeo:**
  - Secuencia de imágenes y audio. Compresión intraframe e interframe.
  - Códecs: H.264, H.265 (HEVC), AV1 (compresión espacial e interframe).
- **Audio:**
  - Muestreo (44,1 kHz CD / 48 kHz vídeo); cuantificación (16/24 bits).
  - Formatos: WAV/FLAC (sin pérdida), MP3/AAC (con pérdida).

### **1.4 Lógica y Operaciones Binarias**

- Aritmética: suma, resta, multiplicación, división en base 2.
- Lógica digital: puertas AND, OR, NOT, XOR.
- Aplicaciones: ALU, circuitos combinacionales.

### **1.5 Detección y Corrección de Errores**

- Bit de paridad (detección simple).
- CRC (Cyclic Redundancy Check).
- Código de Hamming (corrige 1 bit; MEM ECC).
- Reed-Solomon (múltiples errores; CDs, RAID, QR).

### **1.6 Representación en Big Data y Nube**

- Formatos JSON/BSON, Avro, Protobuf, Parquet, ORC.
- Integración en pipelines cloud, microservicios y análisis masivo (Spark, Hadoop).

### **1.7 Comunicación Digital**

- Modelo Shannon-Weaver: emisor, codificador, canal (ruido), decodificador receptor + (Compresión y cifrado)
- Señales digitales vs analógicas.
- Protocolos: TCP/IP, UDP/IP, Ethernet, WebRTC.

### **1.8 Seguridad**

- Hashing: SHA-256 (integridad), bcrypt/Argon2 (contraseñas).
- Cifrado:
  - Simétrico: AES.
  - Asimétrico: RSA, ECC.
  - Cifrado simétrico pero compartiendo la clave con cifrado Asimétrico (SSL)
  - Aplicaciones: HTTPS, VPN, BitLocker, TLS.

### **1.9 Compresión**

- Sin pérdida: Huffman, LZW (ZIP, PNG).
- Con pérdida: JPEG, MP3, H.264 (optimizadas para multimedia).

### **1.10 Conclusión**

- La correcta representación y manipulación de datos es básica en informática.
- Aplica desde circuitos y sistemas embebidos hasta servicios cloud y Big Data.

## **2. PARTE DIDÁCTICA - 1.º DAM – Módulo: Programación**

### **Unidad didáctica: “Estructuras repetitivas aplicadas a la representación y tratamiento de información”**

#### **2.1 Contextualización didáctica**

- Nivel: 1.º de Ciclo Formativo de Grado Superior – Desarrollo de Aplicaciones Multiplataforma (DAM)
- Módulo profesional: Programación
- Currículo aplicable: Real Decreto 450/2010 y Decreto 48/2011 (Comunitat Valenciana)
- **Justificación:** Las estructuras repetitivas permiten automatizar tareas, manipular información y modelar procesos fundamentales en programación profesional.

#### **2.2 Objetivos de aprendizaje**

- Aplicar bucles (for, while, do-while) en la codificación de algoritmos.
- Manipular datos numéricos y textuales mediante estructuras de control repetitivas.
- Simular procesos de codificación, verificación y transmisión de datos.

#### **2.3 Metodología y principios pedagógicos**

- Metodología activa: basada en tareas prácticas y resolución de problemas reales.
- Progresión: ejercicios con dificultad creciente y trabajo individual seguido de refactorización colaborativa.
- Recursos utilizados: IDE Java (NetBeans, VS Code), pseudocódigo, diagramas de flujo, vídeos explicativos.
- Estrategias metodológicas: trabajo por parejas con roles diferenciados, flipped classroom, revisión entre iguales.

#### **2.4 Inclusión y atención a la diversidad (niveles III y IV):**

- Código base parcial con comentarios orientativos.
- Retroalimentación individualizada durante el proceso.

#### **2.5 Aplicación del Diseño Universal para el Aprendizaje (DUA):**

- Múltiples formas de representación (textual, visual, audiovisual).
- Variedad de formas de expresión del aprendizaje (código funcional, exposición oral, demos).
- Participación equitativa mediante ajustes de complejidad y agrupaciones heterogéneas.

#### **2.6 Actividad principal: “Procesando datos binarios con bucles”**

- Conversión manual de números decimales a binario, octal y hexadecimal mediante bucles.
- Codificación de texto carácter a carácter en binario utilizando la tabla ASCII.
- Cálculo de bit de paridad mediante conteo de unos en cadenas binarias.
- Simulación de un canal de transmisión con errores aleatorios y aplicación del Código de Hamming.
- Verificación de integridad de cadenas binarias utilizando un algoritmo de hash simplificado (XOR).

#### **2.7 Evaluación**

- Criterios de evaluación: uso correcto y eficiente de bucles, lógica de control adecuada, limpieza del código, comprensión de los procesos implicados.
- Instrumentos de evaluación: rúbricas detalladas, revisión entre compañeros, evaluación continua con entregas parciales.
- Resultados esperados: desarrollo de programas funcionales que evidencien el dominio de las estructuras repetitivas aplicadas a tareas reales de representación y transmisión de datos.

#### **2.8 Conclusión didáctica**

Esta unidad permite al alumnado integrar conocimientos fundamentales de programación y aplicarlos en situaciones prácticas. Fomenta el pensamiento algorítmico, el desarrollo de habilidades técnicas y competencias transversales, incrementando la motivación y la autonomía. Además, establece conexiones claras entre la teoría informática y su aplicación profesional, desarrollando un aprendizaje significativo.

## Tema 2: Elementos funcionales de un ordenador digital. Arquitectura.

### 1. INTRODUCCIÓN

### 2. ELEMENTOS FUNCIONALES

#### 2.1. Unidad Central de Proceso (CPU - Central Processing Unit)

Encargada de ejecutar instrucciones del programa.

- **Registros:** PC, IR, MAR, MDR, FLAGS.
- **ALU / FPU:** operaciones lógicas y en coma flotante.
- **Unidad de Control:** cableada (rápida) o microprogramada (flexible).

#### 2.2. Memoria principal

Memoria de acceso rápido que almacena temporalmente datos e instrucciones.

- **RAM (Random Access Memory):**
  - **DRAM (Dynamic RAM):** económica, necesita refresco constante.
  - **SRAM (Static RAM):** más rápida, usada en cachés.
- **Jerarquía de memoria:** estructura escalonada que optimiza acceso:
  - Registros > Caché (L1, L2, L3) > RAM > SSD/HDD.
- Afecta directamente al **rendimiento**: menor latencia en niveles superiores.

#### 2.3. Subsistema de Entrada/Salida (E/S)

Permite la interacción del procesador con dispositivos externos.

- **Dispositivos periféricos:** teclado, ratón, impresora, disco, red.
- **Modos de transferencia:**
  - **Polling:** la CPU consulta activamente si hay datos disponibles.
  - **Interrupciones:** el periférico avisa al procesador cuando necesita atención.
  - **DMA (Direct Memory Access):** transfiere datos directamente sin CPU.

#### 2.4. Sistema de buses

Canales físicos que interconectan los componentes del sistema.

- **Tipos:** **Bus de datos** (transmite información), **bus de direcciones** (localiza posiciones de memoria) y **bus de control** (gestiona operaciones como lectura o interrupciones).
- **Temporización:** puede ser **síncrona** (con reloj compartido) o **asíncrona** (mediante señales independientes, más flexible).

### 3. MODELOS DE ARQUITECTURA

#### 3.1. Von Neumann

- Memoria compartida para instrucciones y datos.
- Problema: **cuello de botella** en el acceso a memoria.

#### 3.2. Harvard

- Memoria separada para datos e instrucciones.
- Permite acceso paralelo, más eficiente.

### 4. TAXONOMÍA DE FLYNN

Clasificación de arquitecturas según número de flujos de instrucciones y datos:

- **SISD (Single Instruction, Single Data):** tradicional, una instrucción opera sobre un dato.
- **SIMD (Single Instruction, Multiple Data):** una instrucción actúa sobre múltiples datos (p. ej., GPU).
- **MISD (Multiple Instruction, Single Data):** redundante, escasa utilidad práctica.
- **MIMD (Multiple Instruction, Multiple Data):** múltiples procesadores ejecutan múltiples instrucciones, típico en sistemas multinúcleo.

### 5. MEMORIAS: TIPOS Y EVOLUCIÓN

- **ROM (Read-Only Memory):** no volátil, incluye BIOS/UEFI.
  - **EEPROM:** puede reprogramarse eléctricamente.
- **Flash:** memoria no volátil usada en SSD y dispositivos móviles.
- **Tendencias actuales:**
  - **HBM (High Bandwidth Memory):** gran ancho de banda, muy cercana al procesador.
  - **GDDR6:** memoria gráfica usada en GPUs.
  - **Optane:** tecnología de Intel basada en memoria persistente de alta velocidad.
  - **SoC (System on Chip):** integración total en un único chip, común en móviles.

## 6. CICLO DE INSTRUCCIÓN

Etapas secuenciales que sigue la CPU para ejecutar una instrucción:

1. **Fetch**: se lee la instrucción desde memoria.
2. **Decode**: se interpreta la instrucción.
3. **Execute**: se realiza la operación.
4. **Memory**: acceso a memoria si es necesario.
5. **Write-back**: los resultados se guardan.

Técnicas de optimización:

- **Pipeline**: ejecución en paralelo de etapas.
- **Superescalaridad**: ejecución de múltiples instrucciones por ciclo.
- **Out-of-Order Execution**: reordenamiento dinámico para mejorar rendimiento.
- **SMT (Simultaneous Multithreading)**: varios hilos por núcleo (ej. Hyper-Threading de Intel).
- **Multicore**: varios núcleos físicos en un chip.

## 7. TENDENCIAS FUTURAS

- **Computación cuántica**: uso de **qubits**, permite paralelismo masivo y algoritmos no clásicos.
- **Arquitecturas neuromórficas**: diseñadas para imitar el cerebro humano.
- **Aceleradores de IA**:
  - **TPU (Tensor Processing Unit)**: de Google, optimizadas para redes neuronales.
  - **NPU (Neural Processing Unit)**: en dispositivos móviles.
  - **FPGAs (Field-Programmable Gate Arrays)**: configurables post-fabricación.
- **Sistemas heterogéneos**: combinación de CPU, GPU y otros aceleradores especializados.

## APLICACIÓN DIDÁCTICA (CFGs DAM – Módulo “Programación de procesos y servicios”)

### 1. REQUISITOS PREVIOS

### 2. OBJETIVOS DE APRENDIZAJE

- Analizar el impacto de la arquitectura del sistema en la ejecución de servicios.
- Programar de forma eficiente teniendo en cuenta núcleos, concurrencia y jerarquía de memoria.

### 3. METODOLOGÍA

- ABR (Aprendizaje Basado en Retos). Simulación de entornos reales.
- Uso de herramientas de análisis: **htop**, **perf**, **taskset**, **systemd**.

### 4. ATENCIÓN A LA DIVERSIDAD (Niveles III y IV).

### 5. DISEÑO UNIVERSAL PARA EL APRENDIZAJE (DUA)

- **Representación**: diagramas de arquitectura, vídeos explicativos.
- **Expresión**: scripts, paneles, presentaciones.
- **Compromiso**: retos prácticos contextualizados.

## 6. ACTIVIDAD PRINCIPAL

**“Optimizando procesos según la arquitectura del sistema”** Proyecto práctico por equipos con Python.

Fases:

1. **Análisis del sistema**: detección de arquitectura (núcleos, RAM) con **os**, **platform**, **psutil**.
2. **Programación concurrente**:
  - a. Con **multiprocessing** y **threading**.
  - b. Afinidad a núcleos con **os.sched\_setaffinity()**.
3. **Medición de rendimiento**:
  - a. Scripts instrumentados con **time**, **tracemalloc**.
  - b. Análisis con **perf**, **htop**, comparativa de configuraciones.
4. **Automatización**:
  - a. Scripts como demonios con **systemd**.
  - b. Monitorización de CPU, registro en log.
5. **Defensa y entrega**: Informe técnico + exposición oral con resultados y gráficos.

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

### **1. Introducción**

- La CPU es el núcleo funcional del ordenador: ejecuta instrucciones, coordina operaciones y gestiona recursos.
- Evolución histórica:
  - Mononúcleo: Intel 8086, primeros Pentium.
  - Multinúcleo: Core 2 Duo, AMD Ryzen.
  - Híbridas: Intel Alder Lake, Apple M1/M2 (big.LITTLE).
- Tendencias actuales:
  - Instrucciones específicas para IA (DL Boost, Neural Engine).
  - Integración CPU+GPU.
  - Alta eficiencia energética: clave en móviles y servidores.

### **2. Estructura interna de la CPU**

#### **2.1 Unidad Aritmético-Lógica (ALU)**

- Ejecuta operaciones matemáticas, lógicas y de comparación.
- Registros asociados: acumulador, operandos, flags.
- Unidades especializadas:
  - FPU (coma flotante).
  - SIMD: AVX, SSE, NEON.

#### **2.2 Unidad de Control (UC)**

- Decodifica instrucciones y genera señales de control.
- Tipos:
  - Cableada: más rápida.
  - Microprogramada: más flexible.
- Técnicas modernas:
  - Pipelining.
  - Ejecución fuera de orden y especulativa.
  - Predicción de saltos con IA.

#### **2.3 Memoria interna**

##### **Registros**

- Ultrarrápidos y limitados.
- Generales y especiales: PC, IR, FLAGS, MAR/MDR.

##### **Caché**

- L1: núcleo.
- L2: núcleo o compartida.
- L3: compartida global.
- Técnicas: prefetching, coherencia, reemplazo adaptativo.

##### **RAM**

- Área de trabajo externa.
- DDR5, LPDDR5X según entorno.

#### **2.4 Buses internos**

- Datos, direcciones, control.
- Evolución: FSB → QPI, Infinity Fabric, Unified Memory.

### **3. Funcionamiento de la CPU**

#### **3.1 Conjunto de instrucciones**

##### **CISC**

- Instrucciones complejas.
- Arquitecturas: x86, ARMv8-A.

##### **RISC**

- Instrucciones simples, eficientes.
- Ejemplos: ARM, RISC-V.

##### **Extensiones modernas**

- AVX-512, VT-x/AMD-V, AMX.
- IA integrada, virtualización nativa.
- RISC-V: abierta, modular, en crecimiento.

#### **3.2 Ciclo de instrucción**

- Fases: Fetch → Decode → Execute → Memory Access → Write-back.
- Optimización:



- Multithreading: Hyper-Threading, SMT.
- Predicción, ejecución especulativa.
- Soporte IA en la CPU (Apple Neural Engine, Intel AMX).

#### **4. Conclusión**

Las CPU modernas integran paralelismo, vectores, control avanzado, y capacidades IA. Su comprensión es clave para programar sistemas eficientes, optimizar procesos y entender el funcionamiento base de cualquier equipo digital.

### **PROPUESTA DIDÁCTICA: “SIMULANDO UNA CPU: PROGRAMACIÓN DE UN INTÉRPRETE DE INSTRUCCIONES”**

#### **A. Contextualización**

- Nivel: 1.º FP Grado Superior DAM o DAW.
- Módulo: Programación.
- Perfil: alumnado con dominio básico de estructuras de control y memoria.

#### **B. Objetivos**

- Simular mediante programación el ciclo de instrucción de una CPU.
- Representar digitalmente registros, memoria y operaciones básicas.
- Comprender cómo la CPU gestiona y ejecuta código.

#### **C. Metodología**

- Aprendizaje basado en proyectos y resolución de problemas.
- Desarrollo incremental con pruebas y visualización.
- Programación individual o en parejas (Python, Java, C++).

#### **D. Actividad principal**

- Crear un simulador básico de CPU que:
  - Interprete un pequeño conjunto de instrucciones (LOAD, ADD, JMP...).
  - Emule registros como PC, AC, IR, FLAGS.
  - Visualice el ciclo completo por consola o GUI.
  - Opcional: interrupciones, subrutinas, multithreading.

#### **E. Atención a la diversidad**

- Nivel III: código base, guías, plantillas con instrucciones comentadas.
- Nivel IV: objetivos reducidos, apoyo continuo, evaluación formativa.

#### **F. DUA**

- Representación: consola paso a paso, GUI opcional, esquemas.
- Acción/expresión: elección libre de lenguaje y estructura.
- Implicación: retos progresivos, gamificación por funcionalidades.

#### **G. Evaluación**

- Rúbricas: ejecución correcta, estructura clara, rigor técnico.
- Instrumentos: revisión de código, presentación oral, demo funcional.

#### **H. Conclusión didáctica**

- Programar una CPU permite entender su lógica interna desde el rol de programador.
- Favorece el pensamiento lógico, la abstracción computacional y la transferencia de conocimientos entre hardware y software.

## **Tema 4: Memoria interna: Tipos, Direccionamiento, Características y funciones**

### **1. Introducción**

- La memoria es fundamental para el rendimiento del sistema: almacena instrucciones y datos, afecta a la velocidad de ejecución y coordina con la CPU.
- Una jerarquía eficiente de memoria evita cuellos de botella y maximiza el rendimiento.

### **2. Conceptos fundamentales**

#### **2.1 Elementos clave**

- Soporte físico: silicio (RAM, Flash), magnético (HDD), óptico (CD/DVD).
- Acceso: aleatorio (RAM), secuencial (cintas), asociativo (caché).
- Volatilidad: volátil (RAM), no volátil (Flash, ROM, HDD).

#### **2.2 Direccionamiento**

- 2D: decodificador único, memorias pequeñas.
- 3D: múltiples decodificadores, alto rendimiento.

#### **2.3 Características**

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

### **3. Tipos de memoria**

#### **3.1 Volátiles**

- SRAM: rápida, cara, sin refresco.
- DRAM: necesita refresco, más densa.
- DDR, GDDR, HBM: sincronizadas, especializadas para GPU o IA.

#### **3.2 No volátiles**

- ROM: solo lectura.
- Flash: base de SSD.
- NVRAM: combina velocidad y persistencia.

### **4. Jerarquía de memoria**

- Registros: máxima velocidad, mínima capacidad.
- Caché (L1–L3): latencia reducida.
- RAM: datos activos.
- Almacenamiento (SSD, HDD): persistencia.
- Red/Nube: acceso remoto y respaldo.

### **5. Conexión CPU–Memoria**

#### **5.1 Estructura**

- SRAM: biestables.
- DRAM: condensador.
- ROM: direccionamiento fijo.

#### **5.2 Acceso**

- Buses: direcciones, datos, control.
- Modos: lectura, escritura, modificación.
- Técnicas: paginación, acceso por columnas, refresco.

### **6. Mejora de rendimiento**

#### **6.1 Memoria caché**

- L1–L3 según núcleo o CPU completa.
- Mapeo: directo, asociativo, por conjuntos.
- Reemplazo: LRU, FIFO, aleatorio.

#### **6.2 Memoria virtual**

- **MMU** (Unidad de Gestión de Memoria) convierte direcciones virtuales en físicas mediante
  - **Paginación**: divide la memoria en bloques fijos (páginas y marcos), lo que permite asignación no contigua y evita la fragmentación externa.
  - **Segmentación**: organiza la memoria en bloques lógicos (código, datos, pila), respetando la estructura del programa pero con riesgo de fragmentación externa.
  - **Segmentación** paginada: combina ambos modelos dividiendo cada segmento lógico en páginas, optimizando espacio y manteniendo organización lógica.
- **TLB**: caché de traducciones recientes rápida.

## 7. Tecnologías modernas

- PMEM: persistente como Intel Optane.
- Memoria 3D: mayor densidad, menor latencia.
- PIM: procesamiento en el chip de memoria (IA, HPC).

## PROPUESTA DIDÁCTICA: “SIMULADOR DE JERARQUÍA DE MEMORIA: PROGRAMANDO ACCESOS, LATENCIAS Y CACHE”

### A. Contextualización

- Nivel: 2.º DAM o DAW.
- Módulo: Programación o Sistemas Informáticos.
- Perfil: alumnado con nociones de estructuras de datos y control de flujo.

### B. Objetivos

- Simular la jerarquía de memorias desde registros hasta almacenamiento.
- Comprender cómo la latencia y las políticas de caché afectan al rendimiento.
- Programar comportamientos reales de sistemas modernos de memoria.

### C. Metodología

- Proyecto práctico por parejas o grupos pequeños.
- Enfoque incremental: fases de diseño, implementación, testeo y presentación.
- Lenguajes posibles: Python, Java o C++.

### D. Actividad principal

- Desarrollo de un programa que simule:
  - Memorias con distinta latencia y capacidad (registros, caché, RAM, disco).
  - Políticas de reemplazo de caché (LRU, FIFO, aleatorio).
  - Mapeo directo y asociativo por conjuntos.
  - Acceso secuencial y aleatorio a datos simulados
  - Estadísticas: tasa de aciertos/fallos, tiempo medio de acceso.
- Interfaz: consola o simple GUI para ver operaciones y resultados.
- Ampliación opcional: paginación, uso de TLB y acceso virtual a disco.

### E. Atención a la diversidad

- Nivel III: código base preconfigurado, ayuda estructurada.
- Nivel IV: simulaciones más básicas con componentes seleccionados.

### F. DUA

- Representación: animaciones, diagramas, consola paso a paso.
- Acción: desarrollo libre o guiado, estilos de programación variados.
- Implicación: simulación con ejemplos reales, feedback inmediato.

### G. Evaluación

- Rúbricas: precisión técnica, eficiencia del simulador, claridad del código.
- Instrumentos: demo, documentación del diseño, revisión por pares.

### H. Conclusión didáctica

- Simular memoria refuerza la comprensión del rendimiento real del software.
- El alumnado conecta teoría, arquitectura y programación, desarrollando visión de optimización y eficiencia.

## **Tema 10: Representación interna de los datos**

### **1. Introducción**

- Toda información digital se representa internamente en binario (base 2).
- También se utilizan otras bases para facilitar tareas específicas:
  - Octal (8): sintaxis compacta.
  - Decimal (10): interfaz humana.
  - Hexadecimal (16): dirección de memoria, colores, instrucciones.
- Comprender estas representaciones es esencial en programación, redes y sistemas.

### **2. Representación de caracteres**

- **ASCII**: estándar de 7 u 8 bits.
- **UNICODE (UTF-8, UTF-16)**: codifica miles de símbolos, compatible con la web.
- Conversión texto ↔ binario esencial en programación, bases de datos y redes.

### **3. Representación de booleanos**

- 1 bit: 0 = falso, 1 = verdadero.
- Usos en condiciones, lógica digital, estructuras de control.
- Aplicación en puertas lógicas y simplificación con mapas de Karnaugh.

### **4. Representación de números enteros**

- **Signo y magnitud, CA1, CA2 (complemento a 2)**: estándar en programación.
- **Exceso-Z**: usado en representación de exponentes (coma flotante).

### **5. Representación de reales**

- **Coma fija**: poco precisa.
- **Coma flotante (IEEE 754)**: 32/64/128 bits.
- Componentes: signo, exponente, mantisa.
- Problemas comunes: redondeo, desbordamientos.

### **6. Números complejos**

- Dos flotantes: parte real + imaginaria. Representado con estructura, objeto, etc.
- Usos: audio, señales, simulación física, computación cuántica.

### **7. Representación de estructuras**

#### **7.1 Lineales**

- Vectores, matrices, listas enlazadas: estructuras fundamentales.

#### **7.2 Jerárquicas y grafos**

- Árboles: Árboles: BST (búsqueda binaria), AVL (balanceo estricto), B+ (claves en hojas, ideal para BBDD), R-B (rojo-negro, balanceo por colores).
- Grafos: listas/matrices de adyacencia.
- Aplicaciones en rutas, grafos sociales, IA.

#### **7.3 Tablas hash y punteros**

- Acceso  $O(1)$ , gestión dinámica de memoria.
- Punteros: manipulación directa de direcciones (C/C++).

### **8. Multimedia y datos complejos**

#### **Imagen**

- Raster vs. vectorial.
- Compresión con y sin pérdida (JPEG, PNG).

#### **Sonido y vídeo**

- Muestreo, resolución, formatos (MP3, FLAC, MP4, H.265).
- Codificación por frames (compresión intraframe e interframe), códecs para streaming.

#### **3D**

- Modelado por vértices, formatos: OBJ, GLTF.
- Aplicaciones: videojuegos, simulación física, realidad aumentada.

### **9. Seguridad y compresión**

#### **Cifrado**

- Simétrico, asimétrico, combinación de ambas
- AES, RSA, ECC.
- Criptografía post-cuántica en desarrollo.

#### **Compresión**

- ZIP, PNG (sin pérdida).

- MP3, JPEG (con pérdida).

#### Hash

- SHA, MD5.
- Aplicaciones: autenticación, integridad, búsqueda.

### PROPUESTA DIDÁCTICA: “CREA TU CONVERTOR BINARIO MULTITIPO: EL ORDENADOR DESDE DENTRO”

#### A. Contextualización

- Nivel: 1.º FP DAM o DAW.
- Módulo: Programación.
- Perfil: alumnado con competencias en codificación, estructuras de datos y desarrollo de interfaces.

#### B. Objetivos

- Simular mediante programación la conversión binaria de diversos tipos de datos.
- Visualizar estructuras internas y codificación real.
- Integrar teoría de representación con desarrollo de software funcional.

#### C. Metodología

- Aprendizaje basado en proyectos.
- Desarrollo individual o en parejas.
- Iteración por funcionalidades, pruebas y presentación.

#### D. Actividad principal

##### Proyecto: Programa “BinarioTotal”

- Desarrollo de una aplicación que permita:
  - **Codificar/decodificar:**
    - Caracteres (ASCII, UTF-8).
    - Enteros (CA2).
    - Reales (IEEE 754).
    - Booleanos.
  - **Visualizar:**
    - Codificación interna en binario y hexadecimal.
    - Comparación entre formatos.
  - **Simular estructuras:**
    - Arrays, listas enlazadas, árboles (con impresión en memoria).
  - **Extra** (opcional):
    - Codificar imágenes o sonidos simples.
    - Comprimir o cifrar una cadena o archivo.
- Lenguajes recomendados: Python (Tkinter), Java (Swing/FX), C++ (CLI/GUI simple).
- Interfaz: consola o ventana gráfica básica.

#### E. Atención a la diversidad

- Nivel III: plantillas base, guía paso a paso.
- Nivel IV: estructura modular, evaluación progresiva, refuerzo individual.

#### F. DUA

- Representación: visualización binaria, esquemas gráficos, interfaces.
- Expresión: código, documentación, presentación del proyecto.
- Implicación: simulador personalizado, trabajo en equipo, reto final.

#### G. Evaluación

- Rúbricas: exactitud técnica, calidad del código, visualización, documentación.
- Instrumentos: pruebas funcionales, demo, defensa oral.

#### H. Conclusión didáctica

- El alumnado transforma la abstracción binaria en lógica aplicada.
- Se potencia la comprensión de la arquitectura digital desde el desarrollo de software

## **Tema 11: Organización lógica de los datos. Estructuras estáticas.**

### **1. Introducción**

- Organización lógica: definición abstracta de cómo se agrupan, relacionan y manipulan los datos sin referirse a su almacenamiento físico.
- Importancia: permite diseñar algoritmos eficientes, reutilizables y orientados a tipos.
- Abstracción: separación entre interfaz lógica y estructura física.

### **2. Tipos Abstractos de Datos (TAD)**

- Un TAD define: conjunto de valores, operaciones permitidas y sus propiedades semánticas.
- Ejemplos:
  - **Pila (Stack):** LIFO.
  - **Cola (Queue):** FIFO.
  - **Lista:** secuencia ordenada con inserción/borrado dinámico.
  - **Árbol:** estructura jerárquica padre-hijo.
  - **Grafo:** nodos y aristas, relaciones complejas.
  - **Tabla hash:** acceso  $O(1)$  mediante clave.

### **3. Tipos de datos escalares**

- **Entero:** complemento a 2.
- **Real:** IEEE 754 (simple/doble precisión).
- **Carácter:** Unicode (UTF-8, UTF-16).
- **Booleano:** 0/1.
- **Enumeración:** valores cerrados.
- **Rango:** subset de un tipo base.

### **4. Tipos de datos estructurados**

#### **4.1 Vectores**

- Arrays unidimensionales o multidimensionales para datos indexados.

#### **4.2 Conjuntos**

- Manejo de elementos únicos; operaciones: unión, intersección, diferencia.

#### **4.3 Registros y tuplas**

- Agrupan datos heterogéneos; equivalente a `struct` en C/C++.

### **5. Implementación estática**

#### **5.1 Pilas**

- Array + puntero. Operaciones: `push()`, `pop()`, `top()`. Usos: llamadas, backtracking.

#### **5.2 Colas y deque**

- Array circular; en deque se permite inserción por ambos extremos. Usos: gestión de procesos, buffers.

#### **5.3 Listas**

- Estáticas en array o dinámicas enlazadas; arrays: acceso  $O(1)$ , inserciones costosas; listas: acceso secuencial  $O(n)$ , inserciones eficientes  $O(1)$ .

#### **5.4 Árboles**

- **BST:** elemento izquierdo < nodo < elemento derecho; búsqueda en  $O(\log n)$ , se puede desacoplar.
- **AVL, B+, R-B:** balanceados automáticamente, garantizan  $O(\log n)$ .
- **Heap:** árbol completo implementado en array; usado en colas de prioridad, heapsort.

#### **5.5 Grafos**

- Representación por lista de adyacencia (espacio eficiente) o matriz (acceso rápido). Aplicaciones: rutas, IA, topología.

#### **5.6 Tabla hash**

- Array indexado vía función hash. Colisiones: encadenamiento o direccionamiento abierto. Usos: caché, bases de datos, almacenamiento rápido.

### **6. Utilidad en programación y concursos**

- Las estructuras estáticas son eficientes y predecibles.
- Su estudio es esencial para entrevistas técnicas, competiciones (OIE, ProgramaMe).
- Plataformas de práctica: Codeforces, LeetCode, AtCoder, HackerRank.

## **PROPUESTA DIDÁCTICA: “DISEÑA Y DEFENDE TU ESTRUCTURA: IMPLEMENTACIÓN EN PROGRAMA”**

### **A. Contextualización**

- Nivel: 2.º DAM o DAW.
- Módulo: Programación.
- Perfil: alumnado con dominio de estructuras de control y programación modular.

### **B. Objetivos**

- Identificar el TAD óptimo para resolver un problema real.
- Implementar y defender su estructura mediante código funcional.
- Relacionar su elección con eficiencia y uso de memoria.

### **C. Metodología**

- Trabajo en grupo, enfoque práctico.
- Pasos: análisis → diseño lógico → implementación estática → presentación.

### **D. Actividad principal**

1. Cada grupo selecciona un problema aplicable a una estructura estática (ej. historial, planificación, rutas).
2. Justifica la elección de la estructura (pila, cola, árbol, grafo, hash).
3. Implementa la estructura usando arrays (estática):
  - **Stack**: array + tope.
  - **Queue**: array circular.
  - **BST/AVL/B+**: nodos y punteros.
  - **Heap**: array con operaciones de inserción/eliminación.
  - **Hash Table**: array + función hash + manejo de colisiones.
4. Diseña funciones básicas (insertar, buscar, eliminar, recorrer).
5. Prepara un breve informe en pseudocódigo y organigrama visual.
6. Defiende la solución en una exposición oral, explicando eficiencia y cómo resuelve el problema.
7. Opcional: analiza casos peores y discute mejoras posibles.

### **E. Atención a la diversidad**

- Nivel III: plantillas de estructura y operaciones básicas.
- Nivel IV: planificación por fases, tutoría, roles técnicos dentro del grupo.

### **F. DUA**

- Representación: pseudocódigo, diagramas, esquemas y mapas mentales.
- Expresión: implementación en código, documentación, exposición oral.
- Implicación: cada grupo contextualiza la solución, promueve el debate técnico.

### **G. Evaluación**

- Rúbricas:
  1. Adecuación de la estructura al problema.
  2. Correcta implementación y documentación.
  3. Claridad en la presentación: complejidad, diseño, justificación.
- Instrumentos: observación, demo funcional, revisión del código y autoevaluación.

### **H. Conclusión didáctica**

- Integrar teoría de TADs con implementación estática favorece la comprensión profunda de la programación eficiente.
- El alumnado desarrolla habilidades lógicas, sintácticas y de argumentación técnica.
- La actividad conecta diseño, código y defensa profesional, preparándolos para retos tecnológicos reales.

## **Tema 12: Organización lógica de los datos. Estructura dinámicas.**

### **1. Introducción**

- Las estructuras dinámicas permiten gestionar datos sin tamaño fijo, adaptándose al crecimiento o reducción en tiempo de ejecución.
- Utilizan punteros/referencias para enlazar nodos en memoria y habilitan inserciones y borrados eficientes.

### **2. Fundamentos y características**

#### **2.1 Características clave**

- Asignación de memoria en tiempo real.
- Flexibilidad estructural mediante enlaces.
- Uso intensivo de punteros o referencias.

#### **2.2 Ventajas e inconvenientes**

- **Ventajas:** eficiente en operaciones frecuentes de inserción/borrado.
- **Inconvenientes:** más complejas de implementar, acceso más lento, requerimiento de gestión de memoria manual o automática.

### **3. Listas dinámicas**

#### **3.1 Listas enlazadas simples**

- Nodo: dato + puntero al siguiente.
- Operaciones: insertar, eliminar, buscar, recorrer.
- Ideal para estructuras flexibles.

#### **3.2 Listas doblemente enlazadas**

- Punteros a siguiente y anterior.
- Navegación bidireccional.
- Base para deques o editores de texto.

#### **3.3 Listas circulares**

- El último nodo apunta al primero.
- Útiles en estructuras cíclicas o buffers circulares.

### **4. Pilas y colas dinámicas**

#### **4.1 Pilas (Stack)**

- Implementadas con listas.
- Modelo LIFO: operaciones **push()**, **pop()**, **top()**.
- Aplicaciones: backtracking, llamadas recursivas.

#### **4.2 Colas (Queue)**

- FIFO: operaciones **enqueue()**, **dequeue()**, **front()**.
- Comunes en planificación de tareas.

#### **4.3 Deques**

- Inserción y eliminación por ambos extremos.
- Versátiles para estructuras flexibles.

### **5. Árboles dinámicos**

#### **5.1 Árbol binario**

- Nodo con máximo dos hijos.
- Recorridos: inorden, preorden, postorden.
- Útil en búsqueda y expresión de datos.

#### **5.2 BST**

- Nodo izquierdo < nodo < nodo derecho.
- Búsqueda eficiente (si el árbol está equilibrado).

#### **5.3 Árboles balanceados**

- **AVL y R-B (rojo-negro):** mantienen equilibrio eficiente.
- Mejora de rendimiento en inserciones y búsquedas.

#### **5.4 Árboles n-arios**

- Más de dos hijos por nodo.
- Adecuados para DOM, jerarquías, expresiones múltiples.

#### **5.5 Heap (montículo)**

- Árbol binario completo.
  - **Max-heap:** padres  $\geq$  hijos;
  - **Min-heap:** padres  $\leq$  hijos.
- Implementados en arrays (índices:  $2i+1$ ,  $2i+2$ ).
- Aplicaciones: colas de prioridad, heapsort.

### **6. Grafos dinámicos**

#### **6.1 Listas de adyacencia**



- Cada nodo mantiene lista de vecinos.
  - Eficiente para grafos dispersos.
- 6.2 Nodos enlazados**
- Representan vértices con listas de aristas; permiten grafos dirigidos, ponderados.
- 6.3 Aplicaciones**
- Redes de comunicación, rutas, redes sociales, algoritmos de IA (Dijkstra, A\*).
- 7. Tablas hash con listas encadenadas**
- Utilizan array + función hash.
  - Colisiones resueltas mediante listas enlazadas.
  - Útiles en diccionarios, caches, bases de datos.
- 8. Gestión dinámica de memoria**
- 8.1 Asignación y liberación**
- En C/C++: `malloc/free`, `new/delete`.
  - En Java/Python: recolector de basura automático.
- 8.2 Problemas comunes**
- **Memory leak**: fallos en liberación.
  - **Doble liberación**: errores críticos.
  - **Fragmentación**: ineficiencia en uso de memoria.
- 9. Aplicaciones reales**
- SO: colas de procesos. Compiladores: AST dinámicos.
  - Editores: estructura de líneas. Juegos/simulación: IA y comportamiento dinámico.
  - Bases de datos: índices en árboles B/B+.
- 10. Conclusión**
- Las estructuras dinámicas son esenciales para programas adaptables y funcionales.
  - Su implementación requiere comprensión de punteros, memoria y eficiencia algorítmica.
  - Son fundamentales para sistemas escalables y seguros.

## PROPUESTA DIDÁCTICA: “SIMULA UNA ESTRUCTURA VIVA: PROGRAMA CON NODOS”

### A. Contextualización

- Nivel: 2.º DAM o DAW.
- Módulo: Programación.
- Perfil: alumnado con experiencia en estructuras de control y manejo de memoria.

### B. Objetivos

- Implementar una estructura dinámica en código (lista, árbol, cola, grafo).
- Visualizar su comportamiento (inserción, eliminación, recorrido).
- Relacionar teoría con ejecución dinámica real.

### C. Metodología

- Trabajo por parejas o grupos pequeños.
- Enfoque por fases: elección, diseño, implementación, visualización y exposición.
- Lenguaje recomendado: Java, C++, Python (con GUI opcional).

### D. Actividad principal

1. Seleccionar un caso real (playlist, cola, jerarquía, red).
2. Diseñar la estructura lógica y la interfaz textual o gráfica.
3. Implementar nodos con punteros/referencias y las operaciones básicas:
  - Listas: insertar, eliminar, buscar, recorrer.
  - Árbol: insertar, buscar, balancear, mostrar ordenamientos.
  - Grafos: añadir vértices/aristas, recorrido BFS/DFS.
4. Crear visualización: consola con pasos detallados o GUI sencilla (Tkinter o JavaFX).
5. Presentar ejemplos: cómo cambia la estructura al ejecutar operaciones.
6. Defensoría oral: eficiencia, ventajas y posibles mejoras.

### E. Atención a la diversidad

- Nivel III: estructura base proporcionada, operaciones guiadas.
- Nivel IV: planificación por fases, práctica más autónoma.

### F. DUA

- Representación: diagramas, código interactivo, visualizador de nodos.
- Acción: desarrollo personalizado, elección de casos de uso.
- Implicación: proyecto significativo, defensa colaborativa.

## **Tema 13: Ficheros. Tipos, Características, organizaciones**

### **1.1 Introducción**

- El fichero es la unidad básica de almacenamiento digital.
- Encapsula datos organizados, esenciales para interoperabilidad, seguridad y eficiencia del sistema.
- Ejemplos actuales: ficheros de configuración, logs, multimedia, backups, modelos IA.

### **1.2 Estructura lógica**

- **Campo:** unidad mínima (e.g., "precio").
- **Registro:** conjunto coherente de campos (e.g., ficha de producto).
- **Fichero:** colección organizada de registros.
- Ilustración visual: nombre, precio, categoría → Registro = "Ratón", 25€, "Electrónica".

### **1.3 Tipos de ficheros**

- **Por contenido:** texto plano (.txt), binario (.exe, .png).
- **Por función:** datos (CSV, JSON), scripts (.sh, .py), configuración (.yaml).
- **Por formato moderno:** JSON, Parquet, MP4, FLAC.

### **1.4 Características**

- **Nombre/extensión,** tamaño lógico/físico, **metadatos** (timestamps, permisos).
- Ubicación (ruta), atributos: cifrado, inmutabilidad, enlaces.

### **1.5 Organización física**

Organización	Acceso	Rendimiento	Usos típicos
Secuencial	Lineal	Bajo	Logs
Indexada	Parcial	Medio	Búsquedas
Directa (hash)	Directo	Alto	OLTP, caches
Encadenada	Dinámico	Irregular	Blockchain

### **1.6 Sistemas de archivos**

- **Seguridad:** NTFS (ACL, EFS), APFS, ZFS.
- **Rendimiento:** EXT4, XFS, tmpfs.
- **Distribuidos/Cloud:** S3, HDFS, CephFS.
- **Compatibilidad:** FAT32, exFAT.

### **1.7 Aplicaciones actuales**

- OS: logs, arranque, configuración.
- Bases de datos: ficheros de índices/tablas.
- Web/Móviles: multimedia, JSON.
- IoT, DevOps, IA: binarios, modelos versionados (.pkl, .onnx).

### **1.8 Tendencias**

- Cifrado por defecto (BitLocker, LUKS).
- Clasificación inteligente por IA.
- Cloud-native: sincronización, control de versiones.
- Optimización (S3 lifecycle, edge caching).

### **1.9 Seguridad y versiones**

- Cifrado simétrico (AES-256), asimétrico (PGP).
- Control de versiones: Git, Git LFS, MLflow.
- Ejemplos: backups cifrados, scripts versionados.

### **1.10 Ficheros en la nube**

- Plataformas: Google Drive, S3, Azure Blob.
- Características: accesibilidad, sincronización, versión.
- Riesgos: pérdida de control, mitigación con cifrado y MFA.

### **1.11 Conclusión**

- Los ficheros siguen siendo eje esencial en el almacenamiento moderno.
- Su gestión eficaz garantiza sistemas robustos, seguros y adaptables.

## **2. PARTE DIDÁCTICA**

### **2.1 Contextualización**

### **2.1 Contextualización**

- **Nivel:** Ciclo Formativo de Grado Medio en Sistemas Microinformáticos y Redes (SMR).
- **Módulo:** Servicios en red.
- **Perfil:** alumnado con conocimientos básicos en redes y administración, orientación práctica y profesional.

## 2.2 Objetivos de aprendizaje

- Comprender el papel de los ficheros en los servicios de red.
- Instalar y configurar un servidor FTP y SFTP.
- Gestionar usuarios, permisos, cifrado y estructuras de directorios.
- Valorar la seguridad y la eficiencia en el acceso remoto a ficheros.

## 2.3 Metodología

- **Aprendizaje basado en tareas:** montaje y prueba de un servidor real.
- **Trabajo por proyectos:** simulación de un entorno empresarial.
- Aprendizaje colaborativo: roles técnicos en grupo (admin, usuario, auditor).

## 2.4 Atención a la diversidad (niveles III y IV)

- Desdoblamiento del grupo para tareas prácticas.
- Rúbricas con niveles de logro diferenciados.
- Apoyos visuales y guías paso a paso.
- Tiempo flexible en ejecución de prácticas.

## 2.5 DUA (Diseño Universal para el Aprendizaje)

- Representación: videotutoriales, esquemas, consola práctica.
- Acción: demostración por CLI y GUI (e.g., FileZilla).
- Motivación: práctica conectada con casos reales de empresas.

## 2.6 Actividad principal

**Supuesto didáctico:** "Implementación de un servidor FTP/SFTP seguro para una pequeña empresa"

- El alumnado instalará un servicio FTP en Linux (vsftpd) y lo reforzará con SFTP.
- Configurarán carpetas por usuario, permisos de acceso y autenticación.
- Validarán acceso desde cliente local (FileZilla) y comprobarán transferencia segura.

## 2.7 Evaluación

- **Instrumentos:** rúbricas, observación directa, test técnico.
- **Criterios:**
  - Instalación y configuración funcional.
  - Uso correcto de rutas, permisos y cifrado.
  - Documentación técnica de configuración.
  - Resolución de incidencias simuladas.

## 2.8 Conclusión didáctica

- Comprender los ficheros como elementos clave en servicios de red forma parte del perfil técnico del alumnado SMR.
- La práctica con FTP/SFTP refuerza competencias en seguridad, organización de datos y administración básica de sistemas, aplicables directamente en el entorno laboral.

## Tema 16: Sistemas operativos: Gestión de procesos

## 1.1 Introducción

- Un proceso es un programa en ejecución con recursos del sistema operativo: CPU, RAM, archivos, E/S.
- La gestión eficiente permite multitarea, aislamiento y rendimiento.
- Clave en: servidores cloud, IA, contenedores, microservicios.

## 1.2 Ciclo de Vida del Proceso

- Modelo de 5 estados: Nuevo, Listo, Ejecución, Bloqueado, Terminado.
- Transiciones: dispatch, I/O, interrupciones.

### 1.3 Modelo de 7 Estados (con swap)

- Añade: Listo suspendido y Bloqueado suspendido (disco).
- Mejora la gestión de memoria y escalabilidad.

## 1.4 PCB (Process Control Block)

- Contiene: PID, registros, punteros a memoria, estadísticas, UID/GID.

## 1.5 Hilos (Threads)

- Subprocesos dentro del mismo espacio de direcciones.
- Modelos: 1:1 (Linux), N:1 (limitado), M:N (Go, Erlang).
- Problemas: condiciones de carrera, bloqueos.
- Soluciones: mutex, semáforos, monitores.

## 1.6 Planificación de CPU

- Criterios: turnaround, waiting, response time, throughput.
- Algoritmos: FCFS, SJF, Round Robin, prioridades, multicolos.
- **Linux CFS**: justa distribución con árbol rojo-negro ( $O(\log n)$ ).

## 1.7 Concurrencia y Sincronización

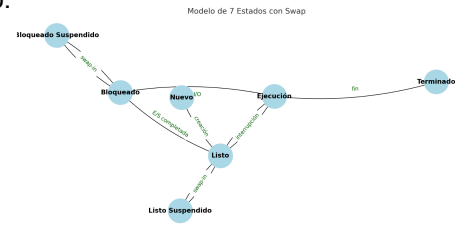
- Sincronización esencial ante acceso compartido a recursos.
- Mecanismos: semáforos (wait/signal), monitores, spinlocks.
- Ejemplo: filósofos comensales.

## 1.8 Comunicación entre Procesos (IPC)

- Técnicas: memoria compartida, pipes, colas, sockets.
- Uso en pipelines de IA, microservicios, servicios distribuidos.

## 1.9 Interbloqueos

- Condiciones de Coffman. Estrategias: prevención, evitación (banquero), detección y recuperación.



### 1.10 Casuística en S.O.

SO	Gestión de procesos	Herramientas
Linux	fork/exec, /proc, CFS	ps, htop, strace
Windows	CreateProcess, scheduler	TaskMgr, PowerShell
Contenedores	namespaces, cgroups	Docker, Podman

## 1.11 Seguridad

- Riesgos: procesos zombie, condiciones de carrera, IPC insegura.
- Mitigación: ASLR, UID, sandbox, auditorías.

## 2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN DE SERVICIOS Y PROCESOS)

## 2.1 Contextualización

- **Nivel:** Grado Superior en DAM/ASIR.
- **Módulo:** Programación de Servicios y Procesos.
- **Perfil:** alumnado con conocimientos en programación y administración de sistemas. Enfoque profesional y automatización.

## 2.2 Objetivos de aprendizaje

- Programar servicios como procesos o demonios.
- Gestionar subprocesos/hilos para tareas concurrentes.
- Implementar sincronización segura entre procesos.
- Utilizar planificación adecuada y comunicación eficiente.

## 2.3 Metodología

- Aprendizaje activo basado en proyectos (ABP).
- Simulación de sistemas reales (servicios Linux, hilos Java o Python).
- Evaluación mediante prácticas funcionales y análisis de código.

## **2.4 Atención a la diversidad (niveles III y IV)**

- Prácticas guiadas con plantillas.
- Tareas escalonadas según nivel.
- Apoyo personalizado en lógica concurrente.

## **2.5 DUA**

- Representación: vídeos, depuradores, visualizadores de procesos.
- Acción: scripting, programación, pruebas automatizadas.
- Compromiso: proyectos aplicados (chat, servidor de colas, servicios REST).

## **2.6 Actividad principal**

### **Supuesto didáctico: “Desarrollo de un servicio concurrente de atención de tareas”**

- Objetivo: desarrollar un servicio en Python/Java que atienda múltiples tareas simulando peticiones de clientes.
- Implementa multihilo, colas de mensajes, sincronización con semáforos.
- Control de estado de procesos y uso de `psutil` (Python) o `Thread` (Java).
- Incluye logs, planificación simple y monitoreo.

## **2.7 Evaluación**

- **Instrumentos:** rúbrica de código, test técnico, checklist de funcionalidades.
- **Criterios:**
  - Correcta creación y control de procesos e hilos.
  - Implementación de sincronización y manejo de condiciones de carrera.
  - Estructura y calidad del código.

## **2.8 Conclusión didáctica**

- La programación y gestión de procesos es esencial para servicios concurrentes.
- Capacita al alumnado para diseñar y mantener software robusto, seguro y eficiente en entornos reales de servidor.
- Este módulo vincula la teoría de los sistemas operativos con la práctica profesional de la programación de servicios.

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

### **1.1. Introducción**

El sistema operativo (SO) es el software fundamental que permite al usuario interactuar con el hardware.

### **1.2. Clasificación de sistemas operativos**

- **Por número de usuarios:**
  - *Monousuario*: un usuario activo por vez. Ej.: Windows Home, macOS.
  - *Multiusuario*: múltiples sesiones simultáneas. Ej.: Linux, Windows Server.
- **Por tareas:**
  - *Monotarea*: obsoleto. *Multitarea*: alternancia (concurrente) o simultaneidad real (paralela).
- **Por núcleo (kernel):**
  - *Monolítico*: alto rendimiento, difícil de mantener (Linux).
  - *Microkernel*: modular, más seguro (Minix, QNX).
  - *Híbrido*: rendimiento y seguridad (Windows NT, macOS).
- **Por arquitectura:**
  - *Sistemas en red*: comparten recursos (FreeBSD, Windows Server).
  - *Distribuidos*: múltiples máquinas como un único sistema lógico (Hadoop, Mesos).
  - *Cloud*: escalabilidad y ejecución bajo demanda (AWS, Azure).
- **Por procesadores:**
  - *SMP (Symmetric Multiprocessing)*: núcleos comparten memoria.
  - *NUMA (Non-Uniform Memory Access)*: varias CPU cada una su propia memoria..
- **RTOS (tiempo real)**: QNX, VxWorks.
- **Emergentes**: SO para IoT (Zephyr, RIOT).

### **1.3. Explotación de sistemas monousuario**

- Instalación del SO, controladores y software inicial.
- Gestión de cuentas locales y configuración básica.
- Actualizaciones automáticas y medidas de seguridad sencillas.
- Ej.: Windows 11 Home, macOS (Time Machine, Gatekeeper, FileVault).

### **1.4. Explotación y administración de sistemas multiusuario**

- Los procesos se ejecutan en modo usuario, accediendo a recursos del sistema mediante llamadas al sistema (syscalls), que transicionan temporalmente al modo kernel para seguridad y estabilidad.
- **Procesos**: múltiples procesos por usuario, planificación (FIFO, RR, prioridades), IPC (pipes, sockets).
- **Memoria**: paginación, segmentación, swapping, espacio separado usuario/kernel.
- **Servicios**:
  - Linux: systemd, daemons, cron.
  - Windows: Task Scheduler, servicios en segundo plano.
- **Almacenamiento**:
  - Sistemas de archivos: NTFS (Windows), EXT4, Btrfs (copy-on-write, snapshots), ZFS (integridad de datos, replicación). Volúmenes: LVM, Storage Spaces.
- **Gestión avanzada**:
  - Linux: sudo, ACLs, journalctl, cgroups.
  - Windows Server: Active Directory, GPOs, RDP, administración remota.

### **1.5. Virtualización y contenedores**

- **Virtualización completa**: VMs con su propio SO. Ej.: VirtualBox, VMware.
- **Paravirtualización**: uso eficiente del hardware. Ej.: Xen, KVM.
- **Virtualización ligera (contenedores)**: comparten kernel del post. Ej.: Docker, LXC. Aislamiento basado en namespaces y cgroups (CPU, RAM, red). Seguridad con AppArmor.
- **Orquestación**:
  - Kubernetes: escalado, balanceo, pods.
  - Helm: gestión de paquetes en Kubernetes.
- **Comparativa**: Aislamiento: VM total, contenedor parcial. Recursos: VM más consumo, contenedor más eficiente. Arranque: VM lento, contenedor rápido.

### **1.6. Seguridad y monitorización**

- **Seguridad:**
  - Control de acceso mediante DAC (Discretionary), MAC (Mandatory) y RBAC (Role-Based), uso de ACLs, autenticación multifactor
  - Cifrado: BitLocker, LUKS, ZFS.
  - Firewalls: iptables, nftables, Windows Defender.
- **Monitorización:**
  - Linux: htop, Prometheus, Grafana.
  - Windows: Sysinternals, Event Viewer.
- Automatización mediante cron, scripts bash/powershell o herramientas de configuración como Ansible y despliegue remoto con Salt Project.

## 2. PARTE DIDÁCTICA

### 2.1. Contextualización

- **Etapas educativas:** Ciclo Formativo de Grado Superior en Administración de Sistemas Informáticos en Red (ASIR). **Módulo profesional:** *Servicios en Red*.

### 2.2. Objetivos de aprendizaje

- Comprender la estructura y funciones de sistemas operativos.
- Distinguir entre entornos monousuario y multiusuario.
- Aplicar procedimientos de instalación, configuración, monitorización y seguridad.
- Desplegar servicios mediante máquinas virtuales y contenedores.

### 2.3. Metodología

- **Aprendizaje basado en proyectos (ABP):** desarrollo de una red empresarial simulada con dos escenarios operativos.
  - **Técnicas didácticas:** simulación de incidencias, resolución por comandos, checklist técnico.
  - **Entornos reales:** VirtualBox, Docker, Ubuntu Server, Windows Server.

### 2.4. Atención a la diversidad y DUA

- **Nivel III:** entornos preconfigurados, uso de scripts automatizados, apoyo visual.
- **Nivel IV:** videotutoriales, interfaz gráfica de administración, guías paso a paso.
- **DUA:** contenidos accesibles por múltiples vías (CLI, GUI, tutoriales), respuestas prácticas y escritas, retroalimentación inmediata.

### 2.5. Actividad principal: “Administra tu sistema”

- **Objetivo:** configurar y comparar un entorno monousuario y otro multiusuario.
  - Instalación de Windows/macOS (monousuario) y Ubuntu Server (multiusuario).
  - Creación de usuarios, servicios, permisos. Despliegue de contenedor Docker con servicio web.
  - Aplicación de políticas de seguridad y uso de herramientas de monitorización.
- **Producto final:** sistema funcional y documentación técnica comparativa.

## **Tema 21: Sistemas informáticos: Estructura física y funcional**

### **1.1 Introducción**

- Sistema informático: conjunto coordinado de hardware, software y comunicaciones para procesar y almacenar datos.
- Evolución: mainframes → PCs → cloud → edge → IA distribuida.
- Relevancia: soporte de servicios críticos en sociedad e industria.

### **1.2 Clasificación y tendencias**

- **Por tamaño:** microcomputadoras, servidores, supercomputadoras.
- **Por arquitectura:** cliente-servidor, embebido, cloud, IoT.
- **Tendencias actuales:**
  - Serverless (AWS Lambda), contenedores (Docker, K8s).
  - Edge Computing, MicroVMs, SoCs.
  - IA distribuida, computación cuántica/neuromórfica.

### **1.3 Impacto social y ético**

- Digitalización: industria 4.0, smart cities, salud digital.
- Desafíos: brecha digital, sostenibilidad (Green IT), sesgos IA.
- Marco normativo: RGPD, soberanía digital, ética computacional.

### **1.4 Arquitectura física**

- **Modelo Von Neumann:** CPU, memoria, E/S, buses.
- **CPU y coprocesadores:** ALU, registros, GPU, TPU.
- **Memoria:** RAM DDR5, caché, SSD NVMe.
- **Periféricos:** entrada/salida clásicos y avanzados (biometría, VR).
- **Redes:** Ethernet, Wi-Fi 6, 5G, SDN.
- **Eficiencia energética:** refrigeración, renovables, CPDs verdes.

### **1.5 Arquitectura lógica y software**

- **Capas:** Hardware → SO → Middleware → Apps.
- **SO:** Linux, Windows, Android, RTOS.
- **Aplicaciones:** ERP, CAD, multimedia, ofimática.
- **Lenguajes:** Python, Java, Rust; herramientas: Git, CI/CD.
- **Virtualización:** VMware, KVM, contenedores (Docker), MicroVMs (Firecracker).
- **Cloud-native:** microservicios, DevOps, escalabilidad horizontal.

### **1.6 Gestión de recursos**

- Planificación de CPU, asignación de memoria, multitarea.
- Técnicas: paginación, segmentación, swapping.
- Monitorización: top, Grafana, logs, métricas.

### **1.7 Seguridad**

- Principios: confidencialidad, integridad, disponibilidad.
- Técnicas: TLS 1.3, backups, autenticación (OAuth2), Zero Trust.
- Amenazas: malware, DDoS, phishing.

### **1.8 Aplicaciones reales**

- Sanidad: historia clínica electrónica, IA diagnóstica.
- Educación: plataformas LMS.
- Industria: IoT, mantenimiento predictivo.
- Transporte: sensores inteligentes, conducción autónoma.

### **1.9 Conclusión**

- Los SI son base tecnológica y social clave.
- Evolución hacia inteligencia, descentralización y eficiencia.
- Retos: ética, sostenibilidad, seguridad.

## **2. PARTE DIDÁCTICA (IMPLANTACIÓN DE SISTEMAS OPERATIVOS / SERVICIOS EN RED)**

### **2.1 Contextualización**

- **Nivel:** SMR / ASIR.
- **Módulo:** Servicios en Red / Implantación de Sistemas Operativos.
- **Perfil:** alumnado técnico con orientación profesional y práctica de infraestructuras reales.

### **2.2 Objetivos de aprendizaje**

- Comprender la estructura física y lógica de un sistema informático.
- Diseñar arquitecturas de red y servicios funcionales.
- Valorar eficiencia, seguridad y escalabilidad.

### **2.3 Metodología**

- Trabajo cooperativo por proyectos reales.



- Simulación en entornos virtuales (VirtualBox, Proxmox, GNS3).
- Uso de esquemas, planificación técnica y roles.

#### **2.4 Atención a la diversidad (niveles III y IV)**

- Itinerarios personalizados (desde simulación básica a automatización).
- Uso de esquemas guiados y rúbricas diferenciadas.
- Recursos adaptados: videotutoriales, soporte escalonado.

#### **2.5 DUA**

- Representación múltiple: visualización de redes y servicios.
- Expresión: presentación oral, diagramas, documentación.
- Compromiso: contexto real (empresa simulada).

#### **2.6 Actividad principal**

##### **Supuesto didáctico: “Diseña la red y servicios de tu empresa”**

- Equipos organizados en grupos diseñan la infraestructura informática de una empresa ficticia (e.g., gestoría, clínica, tienda).
- Planificar:
  - Dirección IP (IPv4 privada).
  - Topología de red (router, switches, servidores).
- Implementar servicios en red (simulados o reales):
  - DHCP, DNS.
  - FTP/Samba.
  - Servidor web o de correo local.
  - Medidas básicas de seguridad: firewall, control de acceso.
- Justificar decisiones y presentar diseño final.

#### **2.7 Evaluación**

- **Instrumentos:** rúbrica de presentación, checklist de servicios, autoevaluación.
- **Criterios:**
  - Coherencia en diseño físico/lógico.
  - Configuración funcional de servicios.
  - Capacidad de análisis y justificación técnica.

#### **2.8 Conclusión didáctica**

- Esta actividad permite al alumnado aplicar de forma integrada sus conocimientos técnicos.
- Conecta teoría (arquitectura de sistemas) con práctica real de diseño de infraestructuras.
- Fomenta la responsabilidad, el trabajo en equipo y la visión profesional.

## **TEMA 22: PLANIFICACIÓN Y EXPLOTACIÓN DE SISTEMAS INFORMÁTICOS**

### **1. PARTE CIENTÍFICA**

#### **1.1 Introducción**

- Un sistema informático combina hardware, software y redes para ofrecer servicios.
- Su planificación, instalación y mantenimiento permiten un funcionamiento seguro, eficiente y continuo.

#### **1.2 Ciclo de vida**

1. Análisis de necesidades (usuarios, servicios, red).
2. Diseño (estructura física y lógica).
3. Instalación (hardware, cableado, conectividad).
4. Configuración de servicios.
5. Explotación y mantenimiento.
6. Actualización o desmontaje.

#### **1.3 Diseño y planificación**

- **Arquitectura:** cliente-servidor, híbrida.
- **Recursos:** número de equipos, servicios necesarios (DNS, DHCP, FTP).
- **Despliegue:** red cableada o inalámbrica, topología estrella.
- **Presupuesto y escalabilidad:** equipos económicos, posibilidad de crecimiento.

#### **1.4 Instalación y condiciones técnicas**

- **Ubicación de equipos:** racks, armarios, espacio ventilado.
- **Conectividad:** switches, routers, puntos de acceso, cableado estructurado.
- **Electricidad y seguridad física:** SAI, organización de cables.

#### **1.5 Configuración de servicios**

- **DNS/DHCP:** asignación automática de IP, resolución de nombres.
- **FTP o Samba:** compartición de archivos en red.
- **Servidor web local:** acceso a contenidos internos.
- Herramientas: Windows Server, Debian, pfSense.

#### **1.6 Explotación y mantenimiento**

- Roles: administrador de red, usuario final.
- Mantenimiento preventivo: revisión de logs, limpieza de hardware.
- Copias de seguridad: backups semanales, en local o en red.
- Monitorización: ping, ipconfig, netstat, herramientas gráficas.

#### **1.7 Seguridad**

- Contraseñas fuertes, actualización de software.
- Cortafuegos (firewall), control de acceso.
- Copias de seguridad y permisos de carpetas.
- Segmentación básica de red si hay invitados o zona Wi-Fi pública.

#### **1.8 Procedimientos de uso**

- Normas internas: uso adecuado de recursos, acceso a carpetas.
- Gestión de usuarios y permisos (usuarios locales, grupos).
- Documentación: esquemas de red, manuales internos.

#### **1.9 Conclusión**

- La correcta planificación y gestión de redes garantiza servicios disponibles, seguros y adaptados.
- Retos comunes: fallos de conexión, seguridad y actualizaciones.

### **2. PARTE DIDÁCTICA (MÓDULO: SERVICIOS EN RED – 2º SMR)**

#### **2.1 Contextualización**

- **Nivel:** 2º SMR.
- **Módulo:** Servicios en Red.
- **Finalidad:** Aprender a diseñar, instalar y mantener una red básica con servicios funcionales.

#### **2.2 Objetivos de aprendizaje**

- Identificar necesidades de red en un entorno real.
- Diseñar una red física y lógica.
- Instalar y configurar servicios de red básicos.
- Aplicar medidas elementales de seguridad y documentación.

#### **2.3 Metodología**

- Aprendizaje basado en proyectos.
- Trabajo por parejas o tríos.
- Uso de simuladores (Packet Tracer) y máquinas virtuales (VirtualBox con Debian/Windows Server).

#### **2.4 Atención a la diversidad (niveles III y IV)**

- Prácticas guiadas paso a paso.
- Plantillas para documentación y configuración.
- Actividades complementarias o adaptadas según ritmo.

#### **2.5 DUA**

- Representación: esquemas visuales, prácticas reales, videotutoriales.
- Acción: documentación, configuración práctica, defensa del proyecto.
- Compromiso: simulación de casos reales (tienda, colegio, oficina).

#### **2.6 Actividad principal**

##### **Supuesto didáctico: “Proyecto técnico: diseña y configura la red de una tienda”**

- Diseño de red para una tienda con 5 PCs, impresora compartida, Wi-Fi para clientes y servidor de archivos.
- Etapas:
  - Plano de red física (cableado, direccionamiento IP).
  - Instalación de DHCP y DNS en un servidor virtual.
  - Configuración de FTP o Samba para compartir recursos.
  - Aplicación de seguridad básica: firewall y permisos.
  - Elaboración de memoria técnica con justificaciones y esquema de red.
  - Presentación final con defensa del proyecto.

#### **2.7 Evaluación**

- **Instrumentos:** rúbricas, listas de verificación, autoevaluación.
- **Criterios:**
  - Red funcional y segura.
  - Servicios correctamente instalados.
  - Documentación clara y esquemas correctos.
  - Trabajo en equipo y resolución de problemas.

#### **2.8 Conclusión didáctica**

- Esta propuesta permite al alumnado integrar teoría y práctica en un proyecto realista.
- Fomenta la autonomía técnica, la cooperación y la responsabilidad en el diseño y gestión de redes.

## TEMA 23: DISEÑO DE ALGORITMOS. TÉCNICAS DESCRIPTIVAS

### 1.1 Introducción

- Algoritmo: secuencia ordenada y finita de pasos para resolver un problema.
- Propiedades: precisión, determinismo, efectividad, fin.
- Aplicaciones: IA, Big Data, videojuegos, ciberseguridad, sistemas embebidos.

### 1.2 Elementos básicos

- **Instrucciones básicas:** asignación, entrada/salida, operadores.
- **Control:** secuencia, condicionales (**if**, **switch**), iteraciones (**for**, **while**).
- Mejora: validación de entradas, evitar redundancias.

### 1.3 Representación de algoritmos

- **Pseudocódigo:** lenguaje intermedio legible sin sintaxis formal.
- **Diagramas de flujo:** visual, útil para planificación inicial.
- **Nassi-Shneiderman:** estructurados, ideales para programación modular.
- **Tablas de decisión:** lógica compleja con múltiples reglas.
- **Entornos visuales:** Scratch, Blockly, App Inventor → ideal para aprendizaje inicial.
- **IA/LLMs:** herramientas como ChatGPT o Copilot generan código y ayudan a depurar.
- **Prototipado interactivo (Figma):** simulan flujo algorítmico de pantallas/interacciones.

### 1.4 Metodología de diseño

- **Análisis:** entradas, salidas, restricciones.
- **Estructuras de datos:** arrays, listas, pilas, árboles.
- **Top-Down (modular):** dividir en funciones.
- **Pruebas:** verificación con casos típicos/extremos.

### 1.5 Técnicas avanzadas

Técnica	Uso común	Ejemplo
Divide y vencerás	Subproblemas independientes	Mergesort, Quicksort
Dinámica	Reutiliza subresultados	Fibonacci, mochila
Greedy (voraz)	Aproximaciones rápidas	Dijkstra, cambio
Backtracking	Exploración completa	Sudoku, N reinas
Evolutivos	Optimización compleja	Algoritmos genéticos

### 1.6 Eficiencia algorítmica

- Notación Big-O: mide tiempo/espacio en función de n.
- Ejemplos:
  - $O(1)$ : acceso array.
  - $O(n)$ : recorrer lista.
  - $O(n \log n)$ : ordenación eficiente.
  - $O(n^2)$ : algoritmos ingenuos.
  - $O(n!)$ : permutaciones.

### 1.7 Aplicaciones prácticas

- Reutilización: ordenación, búsqueda.
- Patrones de diseño: Singleton, Strategy.
- Casos reales: IA (backpropagation), Big Data (MapReduce), RSA (cifrado), Grover (cuántica).

### 1.8 Conclusión

- Pensar algorítmicamente es esencial para cualquier programador.
- Conexión entre problema → lógica → código → interfaz.
- Tendencias: uso de IA, programación visual, computación cuántica.

## 2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN – 2º SMR)

### 2.1 Contextualización

- **Nivel:** 1º DAM.
- **Módulo:** Programación.
- **Perfil:** alumnado en formación inicial, enfocado en resolver problemas lógicos mediante código y representación visual.

## 2.2 Objetivos de aprendizaje

- Comprender qué es un algoritmo y su utilidad práctica.
- Representar soluciones con pseudocódigo, diagramas y entornos visuales.
- Aplicar la lógica algorítmica a tareas reales de programación.
- Relacionar algoritmo → código → experiencia visual (prototipo).

## 2.3 Metodología

- Aprendizaje activo con resolución de problemas contextualizados.
- Secuencia guiada: planteamiento → pseudocódigo → diagrama → código/prototipo.
- Alternancia entre papel, pizarra, herramientas digitales (Scratch, Python, Figma).

## 2.4 Atención a la diversidad (niveles III y IV)

- Actividades escalonadas (resolución básica vs. lógica compleja).
- Soporte visual y verbal: rúbricas, videotutoriales, plantillas de diagramas.
- Ritmos diferenciados en la representación y programación.

## 2.5 DUA

- Múltiples formas de representación (diagrama, código, prototipo).
- Expresión: oral, escrita, gráfica.
- Compromiso: problemas cercanos al alumnado (apps, juegos, decisiones).

## 2.6 Actividad principal

### Supuesto didáctico: “Del problema a la pantalla”

- Fase 1: plantear un problema cotidiano (e.g., calcular descuentos, gestión de turnos, app de reservas).
- Fase 2: representar solución en:
  - Pseudocódigo
  - Diagrama de flujo o Nassi-Shneiderman
- Fase 3: codificar en Python (o Scratch, según nivel).
- Fase 4: diseñar un prototipo en Figma que represente el flujo visual del algoritmo (pantallas, botones, decisiones).

## 2.7 Evaluación

- **Instrumentos:** rúbrica por fases, revisión del código y prototipo, presentación final.
- **Criterios:**
  - Claridad y corrección del pseudocódigo/diagrama.
  - Lógica correcta en la solución.
  - Coherencia entre algoritmo y prototipo visual.
  - Participación activa y comprensión del proceso.

## 2.8 Conclusión didáctica

- El pensamiento algorítmico forma la base de la programación.
- Representar antes de codificar mejora la planificación y reduce errores.
- Vincular lógica + visual + experiencia del usuario refuerza el aprendizaje transversal.

## TEMA 24: LENGUAJES DE PROGRAMACIÓN: TIPOS Y CARACTERÍSTICAS

### 1.1 Introducción

- Lenguaje de programación: sistema formal para expresar algoritmos.
- Actúa como interfaz entre la lógica del programador y la máquina.
- Evolución marcada por las necesidades: eficiencia, mantenibilidad, IA, desarrollo web, etc.

### 1.2 Elementos fundamentales

- **Sintaxis y semántica:** estructura vs. significado.
- **Estructuras de control:** secuencias, condicionales (`if`, `switch`), bucles (`for`, `while`).
- **Tipos de datos:**
  - Primitivos: `int`, `char`, `bool`.
  - Estructurados: arrays, `struct`.
  - Abstractos: listas, pilas, árboles.
- **Tipado:**
  - Estático (C, Java) vs. dinámico (Python, JS).
  - Fuerte (Python) vs. débil (JS).
- Cualidades deseables: legibilidad, seguridad, eficiencia, portabilidad.

### 1.3 Paradigmas de programación

Paradigma	Enfoque	Ejemplo
Imperativo	Cómo hacerlo	C, Python
Declarativo	Qué se desea lograr	SQL
Funcional	Funciones puras	Haskell
Lógico	Reglas y deducción	Prolog
Orientado a objetos	Objetos + métodos	Java, Python
Reactivo	Eventos y reactividad	JavaScript (Vue.js)
Tiempo real	Respuesta inmediata	C embebido
Cuántico	Qubits, superposición	Q#

### 1.4 Clasificación

- **Nivel de abstracción:**
  - Bajo: ensamblador.
  - Medio: C, Rust.
  - Alto: Python, Java.
- **Forma de ejecución:**
  - Compilados: C, C++.
  - Interpretados: Python, JS.
  - Híbridos: Java, C#.
  - Transpilados: TypeScript → JS.
- **Generación:**
  - Clásicos: Pascal, COBOL.
  - Modernos: Kotlin, Go, Rust.
  - Emergentes: Q#, Cirq, Mojo.

### 1.5 Lenguajes y usos

Lenguaje	Usos comunes
C/C++	Sistemas embebidos, drivers
Java	Aplicaciones empresariales, Android

Python	IA, ciencia de datos, automatización
JavaScript	Web frontend/backend
SQL	Bases de datos relacionales
Q#	Computación cuántica

## 1.6 Herramientas

- **Compiladores:** gcc, javac.
- **Intérpretes:** python, node.
- **IDEs:** VS Code, IntelliJ, Replit (cloud).

## 1.7 Tendencias actuales

- Cloud-native (AWS, Azure): Python, Go.
- IA y Machine Learning: Python + TensorFlow.
- No-code/low-code: Appgyver, Glide.
- Programación cuántica: Q#, Cirq.
- Colaboración remota: GitHub Codespaces, Live Share.

## 1.8 Conclusión

- No hay lenguaje universal: depende del objetivo.
- Convivencia de paradigmas.
- Tendencia a herramientas accesibles y entornos colaborativos.

## 2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN – 1º DAM)

### 2.1 Contextualización

- **Nivel:** 1º DAM.
- **Módulo:** Programación.
- **Perfil:** alumnado con conocimientos básicos en algoritmia y estructuras de control, en fase de exploración de distintos lenguajes y entornos.

### 2.2 Objetivos de aprendizaje

- Comprender tipos y características de los lenguajes.
- Identificar sus paradigmas, niveles y aplicaciones.
- Saber elegir el lenguaje adecuado según el contexto.

### 2.3 Metodología

- Comparación activa de lenguajes.
- Representación en pseudocódigo y práctica en entornos reales (Python, JS).
- Trabajo colaborativo: investigación + exposición.

### 2.4 Atención a la diversidad (niveles III y IV)

- Fichas resumen de lenguajes.
- Actividades escalonadas según profundidad técnica.
- Soporte visual (tablas comparativas, infografías).

### 2.5 DUA

- Representación: tablas, código, analogías (lenguaje como perfil laboral).
- Expresión: manifiesto técnico, presentación oral, prototipo funcional.
- Compromiso: formato “debate de lenguajes” + proyecto creativo.

### 2.6 Actividades principales

#### Actividad 1: “Elige tu lenguaje: manifiesto y debate técnico”

- Cada grupo adopta un lenguaje y crea un manifiesto técnico:
  - Paradigma, tipado, ejecución, herramientas, aplicaciones.
  - “Personificación” del lenguaje: defensa como si fuera una entrevista.
- Presentación final en un debate estilo “panel de selección”.

#### Actividad 2: “Diseña tu algoritmo estrella”

- Reto real (ej. gestión de turnos).
- Desarrollo con paradigma estructurado:
  - Pseudocódigo, funciones, estructuras de control.
  - Explicación de cada función como si fuera un “candidato estrella”.
- Simulación del código: Python.
- Evaluación mediante rúbrica compartida (profesor y compañeros).
-

## **TEMA 25: PROGRAMACIÓN ESTRUCTURADA. ESTRUCTURAS BÁSICAS. FUNCIONES Y PROCEDIMIENTOS**

### **1. PARTE CIENTÍFICA**

#### **1.1 Introducción**

- Paradigma que organiza el código de forma lógica y modular.
- Base técnica y pedagógica para aprender POO, desarrollo web o scripting.
- Aporta claridad, facilidad de mantenimiento y depuración.

#### **1.2 Origen y objetivos**

- Propuesta por Dijkstra (1970s) como alternativa al uso de **goto**.
- Favorece código legible, reusable y testeable.
- Beneficios: reducción de errores, mayor control del flujo, colaboración efectiva.

#### **1.3 Estructuras de control**

- **Secuencia:** ejecución lineal de instrucciones.
- **Selección:** decisiones (**if**, **else**, **switch-case**, ternario).
- **Iteración:** bucles (**for**, **while**, **do-while**).
- Ejemplo en Java:

java

```
if (nota >= 5) {  
    System.out.println("Aprobado");  
} else {  
    System.out.println("Suspenso");  
}
```

#### **1.4 Funciones y procedimientos**

- **Funciones:** devuelven valor (**return**), permiten encapsular lógica.
- **Procedimientos:** ejecutan tareas sin retorno.
- **Parámetros:** por valor o referencia, con o sin valores por defecto.
- **Ámbito:** variables locales/globales.
- **Recursividad:** función que se invoca a sí misma (requiere caso base).

#### **1.5 Diseño modular**

- Organización en métodos/clases.
- Reutilización, mantenimiento y pruebas facilitadas.
- Buenas prácticas:
  - Funciones pequeñas, bien nombradas.
  - Uso de **return** para controlar el flujo.

#### **1.6 Control de errores**

- Manejo de excepciones (**try-catch-finally**).
- Validación de entrada y flujo seguro.

java

```
try {  
    int resultado = 10 / divisor;  
} catch (ArithmeticException e) {  
    System.out.println("No se puede dividir por cero");  
}
```

#### **1.7 Eficiencia algorítmica**

- Comparación de soluciones (iterativa vs recursiva).
- Uso de notación Big-O para evaluar rendimiento.
- Técnicas: refactorización, memorización.

#### **1.8 Comparación con otros paradigmas**

Aspecto	Programación Estructurada	POO
Unidad básica	Funciones	Clases/objetos
Uso ideal	Algoritmos, lógica	Aplicaciones complejas
Datos	Acceso directo	Encapsulados

### **2. PARTE DIDÁCTICA (MÓDULO: PROGRAMACIÓN – 1º DAM)**

#### **2.1 Contextualización**



- **Nivel:** 1º DAM.
- **Módulo:** Programación.
- **Perfil:** alumnado en formación inicial en desarrollo de software, que necesita dominar la lógica estructurada antes de pasar a la POO.

## 2.2 Objetivos de aprendizaje

- Comprender y aplicar estructuras básicas.
- Implementar funciones con parámetros y retorno.
- Dividir el código en bloques reutilizables y legibles.
- Evaluar soluciones eficientes con control de errores.

## 2.3 Metodología

- Codificación práctica en Java desde el primer momento.
- Enfoque descendente: problema → algoritmo → código modular.
- Actividades individuales y en grupo con retos progresivos.

## 2.4 Atención a la diversidad (niveles III y IV)

- Proyectos escalables por dificultad.
- Plantillas de código base.
- Seguimiento con feedback personalizado en IDE.

## 2.5 DUA

- Representación: pseudocódigo + código real + diagramas.
- Expresión: defensa de funciones, documentación técnica.
- Compromiso: gamificación, retos competitivos y cercanos.

## 2.6 Actividad principal

### Supuesto didáctico: “Diseña tu algoritmo estrella: el reality show de la eficiencia”

- Fase 1: se asigna un reto cotidiano (gestión de turnos, reservas, control de acceso).
- Fase 2: diseño con estructuras (**if**, **for**, funciones, validación).
- Fase 3: codificación en Java con funciones estructuradas.
- Fase 4: presentación de funciones clave como “candidatos estrella”:
  - Qué hacen, cómo están optimizadas, por qué son útiles.
- Fase 5: ejecución en entorno real (Eclipse o IntelliJ).

## 2.7 Evaluación


- **Instrumentos:** rúbricas, revisión de código, exposición oral.
- **Criterios:**
  - Uso correcto de estructuras de control.
  - Diseño modular, claro y funcional.
  - Código eficiente, comentado y ejecutable.
  - Argumentación sólida de las decisiones tomadas.

## 2.8 Conclusión didáctica

- La programación estructurada enseña a pensar en bloques lógicos.
- Es el primer paso para crear software robusto, mantenible y escalable.
- Sentar estas bases asegura éxito posterior en programación orientada a objetos y desarrollo de aplicaciones complejas.

## TEMA 26: PROGRAMACIÓN MODULAR. DISEÑO DE FUNCIONES, RECURSIVIDAD Y LIBRERÍAS

### 1.1 Introducción a la modularidad

- Divide proyectos en módulos independientes y cohesivos.
- Facilita mantenimiento, reutilización, tests y colaboración.
-  **Anécdota:** fallo de pila por recursión sin caso base—importancia de detener bucles infinitos.

### 1.2 Fundamentos de módulos

**Módulo:** unidad de código con interfaz pública y lógica aislada.

python

```
# modulo_usuario.py
```

```
def registrar(usuario):
```

```
    return True
```

- **Alta cohesión + bajo acoplamiento:**
  - Cohesión: un solo propósito.
  - Acoplamiento: pocas dependencias externas.
- La comunicación se hace por interfaces limpias: parámetros, eventos, retorno.

### 1.3 Diseño de funciones

- Método DRY/SRP: cada función hace una y solo una cosa.
- Parámetros claros ( $\leq 3$ ); usar objetos si son muchos.
- Variables locales frente a globales: evitar efectos invisibles.
- Refactorización: dividir funciones complejas en subrutinas bien nombradas.
- Estructura del proyecto en carpetas (*controllers*, *services*, *utils*).

### 1.4 Recursividad

- Estructura:
  - Caso base: termina la cadena recursiva.
  - Caso recursivo: reduce el problema.

Ejemplo:

python

```
def factorial(n):
```

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

```
    ◦
```

- Optimización: memorización, tail recursion.
- Comparativa:

	Recursiva	Iterativa
Claridad	Buena para árboles	Buena para bucles
Eficiencia (CPU/memoria)	Menor	Mayor
Ideal para	Backtracking, árboles	Repeticiones simples

### 1.5 Librerías y reutilización

- **Estándar vs externas** (pip, npm).
- Documentación (README, docstrings), versionado semántico.
- Empaquetado:
  - Python: `setup.py`, `__init__.py`
  - JavaScript: `package.json`
- Evitar dependencias innecesarias; proteger seguridad.

### 1.6 Modularidad en sistemas reales

- **Microservicios**, **capas lógicas**, **REST/API**, **mensajería** (RabbitMQ), tests unitarios (pytest, Jest).

### 1.7 Conclusión científica

- Modularidad = núcleo del desarrollo profesional.
- Mejora legibilidad, robustez y colaboración.
- Recursividad y librerías son herramientas complementarias esenciales.

## 2. PARTE DIDÁCTICA

### (MÓDULO: PROGRAMACIÓN – 1º DAM)

## 2.1 Contextualización

- **Nivel:** 1º DAM.
- **Módulo:** Programación.
- **Perfil:** futuro desarrollador de software con formación en arquitectura limpia y buenas prácticas de construcción de aplicaciones modulares.

## 2.2 Objetivos de aprendizaje

- Diseñar módulos con alta cohesión y bajo acoplamiento.
- Crear funciones limpias, eficientes y fáciles de mantener.
- Aplicar recursividad de forma controlada.
- Utilizar y construir librerías propias.

## 2.3 Metodología

- Talleres: de monolito a módulos.
- Desarrollo iterativo con herramientas reales (VS Code, Python).
- Revisión de código colaborativa (peer review).

## 2.4 Atención a la diversidad (niveles III y IV)

- Pauta por capas: modularización progresiva.
- Recursos de ayuda: plantillas, tutoriales modular en vídeo.
- Feedback directo en IDE para mejorar diseño.

## 2.5 DUA

- Representación: diagramas de componentes, composición modular.
- Acción: desarrollo de librerías y pruebas.
- Expresión: documentación técnica, defensa de decisiones.
- Compromiso: retos con gamificación (“función estrella”).

## 2.6 Actividad principal

### Concurso “¡Modula y vencerás!”

- **Fase 1:** cada grupo recibe un encargo (e.g., gestión de reservas, validación de formularios, descuentos/prod).
- **Fase 2:** definir módulos (usuarios, reservas, pagos), funciones SRP, lógica de recursividad si procede.
- **Fase 3:** cada equipo presenta “función estrella”: qué hace, eficiencia, por qué modular.
- **Fase 4:** explicar recursividad (o por qué usar iteración).
- **Fase 5:** demostrar ejecución real y presentación de arquitectura modular.

## 2.7 Evaluación

- **Instrumentos:** rúbricas, revisión de código, presentación de la arquitectura.
- **Criterios de valoración:**
  - Claridad y cohesión de los módulos.
  - Calidad de funciones: nombres, cobertura, eficiencia.
  - Recursividad: adecuada y segura (si utilizada).
  - Integración de librerías, buenas prácticas y pruebas unitarias.

## 2.8 Conclusión didáctica

- Fomenta pensamiento arquitectónico desde el inicio.
- Prepara al alumnado para sistemas colaborativos, pruebas y mantenimiento.
- Le da herramientas reales para comenzar proyectos profesionales modulares.

## **Tema 27: Programación orientada a objetos**

### **1.1 Introducción y Motivación**

- La POO modela el mundo real agrupando estado y comportamiento en **objetos**.
- Permite sistemas más escalables y mantenibles que la programación estructurada.

### **1.2 Evolución histórica**

- Pre-POO: C, Pascal – lógica y datos separados.
- Aparición de POO: Smalltalk, C++, Objective-C.
- Consolidación: Java, C#, Kotlin, con gestión automática de memoria.

### **1.3 Ventajas clave**

- **Abstracción:** Clases modelan entidades del dominio.
- **Encapsulamiento:** Protege datos internos.
- **Herencia:** Reutiliza código.
- **Polimorfismo:** Flexibilidad según tipo real.

### **1.4 Clases y Objetos**

- **Clase = molde, Objeto = instancia** (e.g. `Coche miTesla = new Coche()`).
- **Atributos:** de instancia, estáticos, constantes.
- **Métodos:** instancia o estáticos; getters y setters para encapsulado.

### **1.5 Visibilidad y Encapsulamiento**

- Modificadores: `public`, `private`, `protected`.
- Control de acceso: `private int velocidad;` `public void setVel(int v){...}`.

### **1.6 Relaciones entre clases**

- **Asociación, agregación, composición** (diferentes niveles de dependencia).
- **Inyección de dependencias (DI):** reduce acoplamiento; frameworks: Spring, Unity.

### **1.7 Herencia y Polimorfismo**

- **Herencia:** simple, múltiple, multinivel; uso con diligencia.
- **Polimorfismo:**
  - Sobrecarga: mismos métodos con distintas firmas.
  - Sobrescritura: redefinir método en subclase.
  - Enlace dinámico: `Animal a = new Perro(); a.hacerSonido();`.

### **1.8 Principios SOLID**

- S: responsabilidad única.
- O: abierto/cerrado.
- L: sustitución (Liskov).
- I: interfaces específicas.
- D: depender de abstracciones.

### **1.9 Lenguajes y ejemplos**

- **Java:** puramente OO, sin herencia múltiple.
- **C++:** OO con herencia múltiple y control manual.
- **Python:** OO parcial, mixins.
- **C#:** OO puro, interfaces.
- **Ejemplos:** Spring, Entity Framework, Django ORM.

### **1.10 Patrones y arquitecturas**

- **Patrones:** Factory, Decorator, Observer, Strategy.
- **Arquitectura:** MVC, Arquitectura limpia/hexagonal, microservicios.

### **1.11 Conclusión científica**

- La POO es base de ingeniería moderna: modular, reusable y alineada al mundo real.

## **2. PARTE DIDÁCTICA**

### **(MÓDULO: PROGRAMACIÓN – 1º DAM)**

#### **2.1 Contextualización**

- **Nivel:** 1º DAM.
- **Módulo:** Programación.
- **Perfil:** alumnado que debe dominar diseño OO como herramienta profesional.

#### **2.2 Objetivos de aprendizaje**

- Modelar entidades del mundo real con clases.
- Implementar herencia, polimorfismo y encapsulamiento.
- Aplicar principios SOLID y patrones básicos.

#### **2.3 Metodología**

- Desarrollo por fases: análisis → clases → relaciones → código en Java.
- Talleres colaborativos con refactorización y revisión conjunta de código.

#### 2.4 Atención a la diversidad

- Modelos DOM vs API como niveles de complejidad.
- Plantillas base y guías activas.
- IDE con análisis estático (como SonarLint) para reforzar buenas prácticas.

#### 2.5 DUA

- Representaciones: UML, diagramas de clases, ejemplos en código.
- Acciones: codificación, refactorización, test unitarios.
- Motivación: simular trabajo real de arquitecto OO.

#### 2.6 Actividad principal

##### Proyecto “La startup de los objetos”

- Cada grupo define el caso de uso para una startup (e-commerce, reservas, etc.).
- Diseñan el **modelo de dominio** (Usuario, Producto, Pedido, etc.) aplicando herencia y composición.
- Implementan en Java: clases, paquetes, interfaces.
- Explican decisiones de diseño (SOLID, patrones).
- Presentan la arquitectura OO simulando reunión técnica con stakeholders.

#### 2.7 Evaluación

- **Instrumentos:** rúbrica global de diseño y código, exposición grupal.
- **Criterios:**
  - Modelo correcto y coherente.
  - Representación limpia (UML).
  - Implementación funcional y modularizada.
  - Justificación técnica y calidad en la presentación.

#### 2.8 Conclusión didáctica

- La actividad integra diseño, colaboración y comunicación técnica.
- Prepara al alumnado para roles reales como desarrolladores y arquitectos de software.
- Refuerza habilidades interdisciplinarias clave: modelado, codificación, trabajo en equipo, comunicación técnica.

## **Tema 31: Lenguaje C. Estructura, Funciones, Entorno y Depuración**

### **1.1 Introducción y Características**

- Creado por Dennis Ritchie (1972). Base de UNIX.
- Lenguaje de **medio nivel**: control del hardware + abstracción.
- Portabilidad, eficiencia, sintaxis compacta, modularidad.
- Estándares: ANSI C, C99, C11, C17.

### **1.2 Elementos del lenguaje**

- Directivas (**#include**, **#define**), función **main()**, comentarios (**//**, **/\* \*/**).
- Tipos: **int**, **char**, **float**, **double**; modificadores (**long**, **unsigned**).
- Operadores: aritméticos, lógicos, bit a bit, relacionales.

### **1.3 Estructura modular**

- Separación en **.h** (interfaces) y **.c** (implementación).
- Preprocesador: macros, inclusión condicional (**#ifndef**, **#define**).
- Ejemplo:

C

```
// operaciones.h
```

```
int sumar(int, int);
```

```
// operaciones.c
```

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

### **1.4 Funciones**

- Estándar (**stdio.h**, **stdlib.h**, **string.h**, **math.h**): **printf**, **malloc**, **strlen**, **sqrt**.
- Usuario: prototipo en **.h**, definición en **.c**, parámetros por valor o puntero.
- Recursividad: **factorial(n)** como ejemplo clásico.

### **1.5 Punteros y memoria**

- Variables que almacenan direcciones (**int \*p = &a;**).
- Acceso indirecto con **\*p**.
- Memoria dinámica: **malloc**, **calloc**, **realloc**, **free**.
- Errores comunes: memory leaks, segmentation fault, buffer overflow.
- Herramienta: Valgrind.

### **1.6 Proceso de compilación**

- Fases: preprocesado, compilación, enlazado, ejecución.
- Herramientas: **gcc**, **clang**, **make**.

make

```
# Makefile
```

```
programa: main.o operaciones.o
```

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

### **1.7 Depuración y testing**

- **GDB**: puntos de ruptura, ejecución paso a paso, inspección de variables.
- **Valgrind**: errores de memoria.
- **Sanitizers**: **-fsanitize=address**.
- Testing: **assert**, frameworks: Check, Unity.
- Optimización: **-O2**, **-g**, **gprof**.

### **1.8 Comparativa con otros lenguajes**

Lenguaje	Ventajas frente a C	Desventajas
Python	Legible, dinámico	Más lento
Java	Seguro, OOP	Requiere VM
Rust	Memoria segura	Curva alta

### **1.9 Conclusión técnica**

C es esencial en programación de sistemas, embebidos, compiladores y permite comprensión profunda de memoria, eficiencia y arquitectura software.

## **2. PARTE DIDÁCTICA**

### **(MÓDULO: PROGRAMACIÓN – 1º DAM)**

#### **2.1 Contextualización**

- Nivel: 1º DAM.

- **Módulo:** Programación.
- Lenguaje C se emplea para formar bases sólidas en estructuras de datos, gestión de memoria y desarrollo profesional de bajo nivel.

## 2.2 Objetivos de aprendizaje

- Comprender la estructura de programas en C.
- Usar punteros, funciones, compilación modular y librerías.
- Aplicar herramientas de depuración y testing profesional.

## 2.3 Metodología

- Enfoque práctico: codificación diaria, ejercicios dirigidos y proyectos grupales.
- Apoyo visual (diagrama de compilación, árbol de dependencias).
- IDEs ligeros: Code::Blocks, Geany + consola.

## 2.4 Atención a la diversidad

- Plantillas con código base comentado.
- Pares tutores para seguimiento.
- Ajustes en número de funciones/módulos según nivel (III y IV).

## 2.5 DUA

- **Representación:** UML, código comentado, ejecución paso a paso.
- **Acción:** prácticas con funciones, Makefile, GDB.
- **Motivación:** desafíos entre equipos tipo “debugging hunt”.

## 2.6 Actividad principal

### “C-Reto: construye tu microkernel modular”

- Grupos de 2-3 alumnos simulan construir un sistema embebido.
- Módulos: **entrada**, **lógica**, **salida**, **errores**, **depuración**.
- Uso de **.h/.c**, punteros, **malloc**, **assert**, Makefile, Valgrind.
- Fase final: defensa técnica + demo + validación de errores.

## 2.7 Evaluación

- **Instrumentos:** rúbrica modular + validación técnica + defensa oral.
- **Criterios:**
  - Código limpio, modular, compilable.
  - Uso adecuado de funciones y punteros.
  - Depuración funcional con herramientas.
  - Claridad en la defensa técnica del diseño.

## 2.8 Conclusión didáctica

El lenguaje C proporciona al alumnado un dominio técnico esencial sobre la estructura y el funcionamiento de los programas. Su estudio práctico, mediante retos modulares y herramientas profesionales, potencia habilidades clave en desarrollo software profesional y prepara para materias futuras como Sistemas, Bases de Datos y Programación Orientada a Objetos.

## **Tema 32: Manipulación de estructuras en C. Punteros, funciones**

### **1.1 Introducción**

- C ofrece **control total sobre memoria**, estructuras y funciones.
- Ideal para aprender cómo funciona la memoria, la modularización y la lógica algorítmica avanzada.

### **1.2 Estructuras estáticas**

- **Arrays**: homogéneos, acceso rápido por índice, tamaño fijo.
- **struct**: agrupación de campos heterogéneos (**Alumno**).
- **union**: campos que comparten memoria (eficiencia).
- **enum**: constantes simbólicas legibles (**enum Estado {ACTIVO, INACTIVO}**).

### **1.3 Estructuras dinámicas**

- **malloc, calloc, realloc, free**: gestión manual de memoria.
- **Listas enlazadas**: nodos dinámicos con punteros.
- **Árboles binarios**: búsqueda ordenada por recursividad.
- **Pilas y colas**: LIFO/FIFO, punteros cabeza/cola.
- **Tablas hash**: dispersión eficiente de claves.
- **Grafos**: listas de adyacencia con punteros.

### **1.4 Entrada y salida**

- E/S estándar: **scanf, fgets, getchar**.
- E/S con archivos: **FILE\* f = fopen(...), fgets, fprintf, fclose**.

### **1.5 Punteros**

- **&** (dirección), **\*** (desreferencia).
- Modificación de variables desde funciones (**void cambiar(int \*p)**).
- **Punteros dobles**: matrices dinámicas, listas dobles.

### **1.6 Punteros a funciones**

- Declaración: **int (\*pf)(int, int) = suma;**
- Uso en menús (**void (\*menu[])() = {ver, editar};**)
- **Callbacks**: funciones pasadas como argumentos (**qsort** con **comparar**).

### **1.7 Modularización**

- **.h** para declaraciones, **.c** para implementación.
- **Makefile** para automatizar compilación.

make

CC=gcc

OBJ=main.o lista.o

prog: \$(OBJ)

\$(CC) -o prog \$(OBJ)

### **1.8 Testing y depuración**

- **assert** para validaciones en ejecución.
- **Valgrind**: fugas, errores de memoria.
- **GDB**: depuración paso a paso.
- **CMocka**: testing profesional, automatizado con mocks.

### **1.9 Buenas prácticas**

- Inicializar punteros, verificar **malloc**.
- Documentar responsabilidad de liberar memoria.
- Separar lógica, E/S y validación.

## **2. PARTE DIDÁCTICA**

### **(MÓDULO: PROGRAMACIÓN – 1º DAM)**

#### **2.1 Contextualización**

- **Nivel**: 1º DAM.
- **Módulo**: Programación.
- El uso de estructuras dinámicas y punteros entrena la lógica algorítmica, la organización modular y la gestión avanzada de memoria.

#### **2.2 Objetivos de aprendizaje**

- Manipular estructuras de datos dinámicas y estáticas en C.
- Usar punteros y funciones para modularizar programas.
- Validar programas con herramientas de testing profesional.

#### **2.3 Metodología**

- **Aprendizaje por proyectos**: desarrollar sistemas pequeños con listas, árboles o colas.
- Prácticas guiadas de implementación, pruebas y depuración.



- Simulación de problemas reales: menú de opciones, carga desde fichero, etc.

#### 2.4 Atención a la diversidad

- Desdoblamiento de funciones: versión básica vs. avanzada.
- Guías paso a paso + comentarios orientativos.
- Evaluación formativa progresiva, con seguimiento individual.

#### 2.5 DUA

- **Representación:** diagramas de estructuras dinámicas, videos paso a paso.
- **Acción:** implementación modular, menús dinámicos, debugging gamificado.
- **Motivación:** retos entre grupos, “modo cazador de bugs”.

#### 2.6 Actividad principal

##### “Misión: estructura viva – modela, enlaza y ejecuta”

- **Fase 1:** diseño en papel/UML de la estructura elegida (lista, pila, árbol...).
- **Fase 2:** codificación modular con `.h/c`, `malloc`, E/S y punteros dobles.
- **Fase 3:** menú dinámico con punteros a funciones.
- **Fase 4:** testing con `assert`, Valgrind y CMocka.
- **Fase 5:** demo técnica con defensa oral del diseño y herramientas usadas.

#### 2.7 Evaluación

- **Instrumentos:** rúbrica técnica + validación funcional.
- **Criterios:**
  - Estructura de datos funcional, uso correcto de memoria.
  - Menú modularizado con punteros a funciones.
  - Uso efectivo de herramientas (Valgrind, `assert`...).
  - Defensa clara y técnica del código.

#### 2.8 Conclusión didáctica

Este tema desarrolla en el alumnado competencias fundamentales para el desarrollo profesional: diseño eficiente de estructuras, modularidad, control total de la memoria y depuración avanzada. Su dominio es esencial como base para asignaturas futuras (ED, Programación OO, Acceso a Datos).

## **Tema 34: Sistemas Gestores de Bases de Datos. Funciones, componentes, arquitecturas y tipos**

### **1.1 Introducción y funciones clave**

- Un SGBD es un sistema de software que administra datos de forma eficiente, segura e integrada.
- Funciones básicas: almacenamiento, consulta, actualización, borrado, control de concurrencia, recuperación ante fallos, seguridad, integridad, respaldo y auditoría.

### **1.2 Componentes del SGBD**

1. **Motor de almacenamiento:** gestiona acceso físico a datos, índices, transacciones, cachés.
2. **Procesador de consultas:** parseo, optimización y ejecución.
3. **Gestor de transacciones:** ACID (atomicidad, consistencia, aislamiento, durabilidad), MVCC, manejo de bloqueos.
4. **Catálogo/diccionario de datos:** metadatos, esquemas, permisos.
5. **Interfaz de usuario:** SQL, GUI, API, drivers.
6. **Módulo de recuperación:** registros de transacción, journaling, restore.

### **1.3 Arquitecturas de referencia**

- **Monolítico:** todo en un solo proceso (SQLite).
- **Cliente-servidor:** cliente envía consultas al servidor (PostgreSQL, MySQL).
- **Distribuido:** datos segmentados en nodos remotos con consistencia distribuida (Cassandra, CockroachDB).
- **En la nube:** SGBD gestionados (AWS RDS, Azure SQL, MongoDB Atlas).

### **1.4 Arquitectura operacional**

- **Capa física:** bloques, ficheros, página de disco.
- **Capa almacenamiento:** buffer cache, índices, tablas.
- **Capa lógica:** vistas, esquema conceptual.
- **Capa de acceso:** SQL, APIs de drivers, ORM.

### **1.5 Tipos de SGBD**

- **Relacionales (SQL):** MySQL, PostgreSQL, Oracle, SQL Server, SQLite.
- **NoSQL:**
  - Documental (MongoDB, CouchDB)
  - Clave-valor (Redis, Memcached)
  - Columnar (Cassandra, HBase)
  - Grafos (Neo4j, JanusGraph)
- **NewSQL:** CockroachDB, Google Spanner (SQL + escalabilidad cloud).
- **Sistemas embebidos:** SQLite, Berkeley DB.

### **1.6 Comparativa**

- **SQL:** ACID, joins, cálculo complejo.
- **NoSQL:** escalabilidad, rendimiento, esquema flexible.
- **Distribuidos/NewSQL:** escalado horizontal + consistencia.
- **Selección depende de:** consistencia, rendimiento, coste, tipo de datos, escalabilidad.

### **1.7 Conclusión técnica**

- Un SGBD moderno es un sistema complejo, con múltiples capas de responsabilidad.
- Su elección y configuración impactan directamente en rendimiento, disponibilidad, consistencia y coste operativo.

## **2. PARTE DIDÁCTICA**

### **(MÓDULO: PROGRAMACIÓN – 1.º DAM)**

#### **2.1 Contextualización**

- El alumnado usa SGBD en prácticamente todos los módulos: backend web, móvil, sensores, microservicios y Big Data.
- Imprescindible dominar estructura interna, funciones y tipos de SGBD.

#### **2.2 Objetivos de aprendizaje**

- Reconocer y describir las funciones internas del SGBD.
- Identificar sus componentes y capas operacionales.
- Comparar tipos según arquitectura y uso.
- Tomar decisiones adecuadas según requisitos de proyecto.

#### **2.3 Metodología**

- Clases teórico-prácticas con esquemas visuales.
- Estudio de casos reales (PostgreSQL, MongoDB, Cassandra).
- Ejercicios comparativos de sistemas y configuraciones.

## 2.4 Atención a la diversidad

- Textos divididos por niveles de profundidad.
- Fases diseñadas: básico (SQL relacional), intermedio (NoSQL documental), avanzado (NewSQL/distribuido).
- Tutorías específicas para aclarar conceptos complejos.

## 2.5 DUA

- Representación: diagramas capa-función.
- Acción: configuración real de instancias locales, monitorización, ejecución de consultas.
- Motivación: simular decisiones técnicas reales en selección/configuración de SGBD.

## 2.6 Actividad principal

### “Comparativa real de SGBD”

1. Grupos simulan una empresa con caso de uso (e-commerce, IoT, reporting...).
2. Diseñan y despliegan tres soluciones:
  - Relacional (PostgreSQL)
  - NoSQL documental (MongoDB)
  - Distributed/NewSQL (CockroachDB o similares)
3. Evaluación de rendimiento: consultas insert/update/select masivas, replicación, fallos simulados.
4. Presentación de conclusiones técnicas: ventajas, limitaciones, elección tomada.

## 2.7 Evaluación

- **Instrumentos:** rúbrica técnica, informe al estilo “technical review”, exposición.
- **Criterios:**
  - Cobertura de funciones internas y arquitecturas.
  - Comparativa argumentada de sistemas.
  - Resultados experimentales y análisis de rendimiento.
  - Claridad en exposición y rigor técnico.

## 2.8 Conclusión didáctica

Este tema dota al alumnado de las habilidades necesarias para comprender y elegir SGBD en entornos reales. Le proporciona una visión integral desde el diseño, la tecnología y la operación, aportando una base sólida para desarrollar e integrar sistemas de información robustos.

## **Tema 35: La definición de datos. Niveles, lenguajes y diccionario**

### **1.1 Introducción**

- Definir datos implica describir **qué** se almacena y **cómo**, clave para la integridad y usabilidad de bases de datos.
- Desde perspectiva de diseño y ejecución, los datos deben documentarse, estructurarse y controlar su uso.

### **1.2 Niveles de descripción (arquitectura ANSI/SPARC)**

1. **Nivel conceptual** – vista global del sistema, con entidades, relaciones y restricciones.
2. **Nivel lógico** – modelo específico: tablas/columnas (SQL) o colecciones/atributos (NoSQL).
3. **Nivel físico** – implementación: ficheros, índices, particiones, rutas en disco.

### **1.3 Modelos y lenguajes de definición**

- **SQL-DDL**: **CREATE TABLE**, tipos (**INT**, **VARCHAR**), **CONSTRAINT** (PK, FK, UNIQUE), **ALTER**, **DROP**.
- **NoSQL-DDL**: definidos en documentos, esquemas flexibles (ej. MongoDB con validadores JSON schema).
- **DDL orientado a objetos**: definición de clases y atributos con herramientas ORM.

### **1.4 Diccionario de datos**

- Catálogo central que describe cada elemento: nombre, tipo, longitud, valores permitidos, significado, propietario, restricciones, etiquetas, origen.
- Permite coherencia semántica, gobernanza y entendimiento compartido.
- Suele incluir metadatos: fechas, responsables, nivel de confidencialidad y ejemplos.

### **1.5 Metadatos adicionales**

- Trazabilidad: quién y cuándo creó/modificó datos.
- Uso: campos obligatorios/opcionales, clasificaciones, relaciones cruzadas.
- Soporta auditoría, control de calidad y seguridad.

### **1.6 Uso en ingeniería de datos**

- Diseño común entre aplicaciones heterogéneas, APIs, servicios, microservicios.
- Base para generación automática de código, validación de datos y documentación interactiva.

### **1.7 Conclusión técnica**

- La definición formal de datos garantiza interoperabilidad, calidad y mantenimiento eficiente de sistemas.
- Hace del diccionario un artefacto clave en proyectos reales de software y bases de datos.

## **2. PARTE DIDÁCTICA**

### **(MÓDULO: PROGRAMACIÓN – 1.º DAM)**

#### **2.1 Contextualización**

- El objetivo es que el alumnado entienda cómo estructurar y documentar datos de forma profesional, habilidad esencial para desarrollo de aplicaciones y servicios.

#### **2.2 Objetivos de aprendizaje**

- Aprender los tres niveles de descripción de datos.
- Usar lenguajes DDL en SQL y ejemplos NoSQL.
- Crear y mantener un diccionario de datos colaborativo.

#### **2.3 Metodología**

- Desarrollo de una base de datos pequeña (ej. tienda online, registro académico).
- Trabajo por fases:
  1. Diagrama entidad-relación (nivel conceptual).
  2. Generación de scripts DDL (nivel lógico).
  3. Implementación de índice, particiones, replicación (nivel físico).
- Registro sistemático de campos en un diccionario compartido (Hoja de cálculo o herramienta colaborativa).

#### **2.4 Atención a la diversidad**

- Plantillas progresivas: de básico (nombre, tipo) a completo (restricciones, origen, nivel confidencial).
- Ajuste de metas según capacidad técnica (niveles III y IV).
- Tutoría individualizada en elaboración de diccionario.

#### **2.5 DUA**

- Representaciones gráficas (ERD, tablas).
- Acción: práctica de definición y documentación de datos.

- Motivación: simular trabajo real de analista/arquitecto de datos.

## **2.6 Actividad principal**

### **“Analiza y documenta tu base”**

1. Fase 1 – modelado conceptual con ERD.
2. Fase 2 – scripts SQL/DDI y validadores NoSQL.
3. Fase 3 – compilación de diccionario: nombre, tipo, longitud, restricciones, descripción.
4. Fase 4 – sugerencia de control físico (índices) y configuraciones.
5. Fase 5 – presentación oral: explicación de elecciones técnicas y convenciones de nomenclatura.

## **2.7 Evaluación**

- **Instrumentos:** rúbrica para evaluación del ERD, scripts, diccionario y exposición.
- **Criterios:**
  - Completitud y coherencia semántica del diccionario.
  - Calidad de scripts DDL y justificación técnica.
  - Claridad en la exposición y capacidad argumentativa.

## **2.8 Conclusión didáctica**

El trabajo sistemático en diseño y documentación de datos implica responsabilidad técnica y facilita futuras integraciones y mantenimiento. El dominio de los tres niveles y el diccionario de datos prepara para roles reales como desarrollador, analista o arquitecto de datos.

## **Tema 36: La manipulación de datos: Modelado, Consultas y Optimización**

### **1.1 Introducción**

- Esencial en cualquier SGBD: almacenar, consultar, actualizar, borrar datos de forma eficiente.
- Base para aplicaciones web, móviles, IoT y Big Data.
- El rendimiento depende del diseño de consultas y estructura de datos.

### **1.2 Modelos de datos**

Modelo	Ejemplo	Características
Relacional (SQL)	Tablas + claves → <b>CREATE TABLE clientes (...)</b>	Normalización, integridad referencial
Documental (NoSQL)	MongoDB con documentos JSON	Agilidad, esquemas flexibles
Clave-valor	Redis cache	Alta velocidad, ideal para sesiones
Columnares	Cassandra	Esquemas densos por columnas
Grafos	Neo4j	Relaciones complejas (social, rutas)
Multimodelo	ArangoDB	Combina documentos, grafos y relaciones
NewSQL	CockroachDB	SQL distribuido con ACID

### **1.3 Lenguajes de manipulación**

- **SQL:**
  - DML: **SELECT, INSERT, UPDATE, DELETE**
  - DDL: **CREATE, ALTER, DROP**
  - DCL: **GRANT, REVOKE**
  - TCL: **BEGIN, COMMIT, ROLLBACK**
- **NoSQL:**
  - MongoDB (MQL): **db.clientes.find({ ... })**, agregaciones
  - Cassandra (CQL): consultas simples por partición
- **SQL con JSON:**
  - PostgreSQL: **SELECT datos->'nombre' FROM empleados WHERE datos->'pais' = 'España';**

### **1.4 Operaciones de datos**

- **Básicas:** **SELECT, INSERT, UPDATE, DELETE**
- **Avanzadas:**
  - **JOIN**, subconsultas, funciones agregadas (**GROUP BY, HAVING**)
- **Transacciones:** ACID, aislamiento, MVCC

### **1.5 NoSQL**

- MongoDB: inserciones y agregaciones eficientes
- Cassandra: consultas clave-primaria, sin JOINS

### **1.6 Optimización de consultas**

- **SQL:**
  - Índices: B-Tree, Hash, GIN
  - EXPLAIN/ANALYZE
  - Reescritura de consultas (evitar **SELECT \***, subconsultas invalidas...)
  - ⚠ caso típico: evitar producto cartesiano mediante subconsulta
- **NoSQL:** índices compuestos, TTL, diseño con enfoque “query-first”
- **Herramientas:**
  - PostgreSQL: **pg\_stat\_statements, auto\_explain**

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

### 1.7 Configuración y escalabilidad

- Ajustes: caches, buffer sizes, conexiones
- Alta disponibilidad: replicación, sharding, tolerancia a fallos
- En la nube: RDS, MongoDB Atlas, escalado gestionado

### 1.8 Tendencias

- Bases autogestionadas (IA integrada)
- Enfoque multimodelo
- Integración con Big Data (Spark, Kafka)
- Columnar para analítica masiva (ClickHouse)

### 1.9 Conclusión técnica

- Modelar datos, saber manipularlos y optimizar consultas garantiza sistemas robustos.
- SQL y NoSQL son complementarios en arquitecturas modernas.

## 2. PARTE DIDÁCTICA

### (MÓDULO: PROGRAMACIÓN – 1.º DAM)

#### 2.1 Contextualización

- **Aplicación** de conceptos en desarrollo real de aplicaciones web y móviles.
- Permite entender impacto del diseño de datos y rendimiento.

#### 2.2 Objetivos de aprendizaje

- Diseñar modelos relacionales y NoSQL.
- Escribir y optimizar consultas SQL y MQL.
- Aplicar índices, revisando planes de ejecución.
- Comparar ventajas y limitaciones de cada enfoque.

#### 2.3 Metodología

- Talleres con bases reales: SQLite, PostgreSQL y MongoDB local
- Ejercicios progresivos: de consultas simples a operaciones complejas.
- Análisis de caso y reescritura de consultas subóptimas.

#### 2.4 Atención a la diversidad

- Niveles diferenciados: grupos en esquema relacional, documental, híbrido.
- Enfoque práctico con guías paso a paso.
- Laboratorio accesible para resolución independiente.

#### 2.5 DUA

- Representación: diagramas de entidad-relación, sketch de documentos JSON
- Acción: modelado y pruebas de consultas, optimización, monitorización
- Compromiso: realizar pruebas en tiempo real del impacto en rendimiento

#### 2.6 Actividad principal

##### Proyecto “Diseña tu Base de Datos del Mundo Real”

- Grupos eligen un dominio (tienda en línea, red social, etc.)
- Diseñan modelo: tablas SQL + documentos NoSQL
- Implementan operaciones CRUD, agregaciones y transacciones/multi-documento
- Optimizan con índices, EXPLAIN, reestructuran consultas
- Presentan resultados y justificación de decisiones

#### 2.7 Evaluación

- **Instrumentos:** rúbrica técnica + presentación del proyecto
- **Criterios destacables:**
  - Calidad del modelo y escalabilidad
  - Eficiencia de consultas y uso de índices
  - Entendimiento de la configuración y entorno seleccionado
  - Comunicación técnica clara y coherente

#### 2.8 Conclusión didáctica

Este tema forma al alumnado en el manejo competente de datos, combinando teoría de bases, consultas avanzadas, modelos alternativos y visión de arquitecturas modernas. Les dota de herramientas clave para desarrollar servicios eficientes y escalables.

## Tema 38: Modelo de datos relacional. Estructuras, operaciones y álgebra relacional

### 1.1 Introducción

- Propuesto por **E. F. Codd (1970)**. Representa los datos como **tablas** (relaciones) con **tuplas** (filas) y **atributos** (columnas).
- Fundamentos: **teoría de conjuntos y lógica de primer orden**.
- Aporta independencia entre el **modelo lógico** y el **físico**.
- Introduce claves primarias (PK) y foráneas (FK) para mantener integridad referencial.

### 1.2 Estructura de datos

- **Relación**: conjunto de tuplas.
- **Atributo**: columna con dominio de valores.
- **Tupla**: fila que representa un registro.
- **Dominio**: tipo de dato permitido.
- **Esquema**: nombre + atributos.

#### Restricciones:

- Dominio, NOT NULL, UNIQUE, CHECK
- **Integridad de entidad**: PK no nula ni repetida.
- **Referencial**: FK debe existir en la tabla referida.

### 1.3 Operaciones del álgebra relacional

- **Selección ( $\sigma$ )**: filas por condición  $\rightarrow \sigma_{ciudad='Madrid'}(clientes)$
- **Proyección ( $\pi$ )**: columnas  $\rightarrow \pi_{nombre,email}(clientes)$
- **Unión ( $\cup$ )**, **Intersección ( $\cap$ )**, **Diferencia ( $-$ )**
- **Producto cartesiano ( $\times$ )**
- **JOINS**:
  - $\theta$ -join, equi-join, natural join ( $\bowtie$ )

### 1.4 Álgebra relacional y SQL

*El álgebra relacional es un lenguaje formal, y SQL es su implementación práctica.  
Comprender álgebra permite razonar sobre consultas SQL antes de codificarlas.*

Álgebra	SQL
$\sigma$	WHERE
$\pi$	SELECT columnas
$\bowtie$	JOIN ON condición
$\cup$	UNION
$-$	EXCEPT / MINUS
$\div$	Subconsultas correladas

#### Ejemplo:

sql

```
SELECT nombre FROM clientes  
JOIN pedidos ON clientes.id = pedidos.cliente_id  
WHERE pedidos.total > 100;
```

Álgebra:  $\pi_{nombre}(clientes \bowtie \sigma_{total>100}(pedidos))$

### 1.5 Normalización

- **1FN**: sin atributos multivaluados.
- **2FN**: sin dependencias parciales.
- **3FN**: sin dependencias transitivas.  
 $\rightarrow$  Evita redundancias, mejora integridad.

### 1.6 Perspectiva actual

- Modelo relacional sigue vigente: MySQL, PostgreSQL, Oracle.
- Adaptación a Big Data, nube, NewSQL, bases multimodelo.



- Complementado por modelos NoSQL según contexto.

### 1.7 Conclusión técnica

- Modelo robusto, lógico, estándar.
- Álgebra relacional: base para formular, depurar y optimizar consultas.
- Conocimientos esenciales para desarrollo profesional y sistemas reales.

## 2. PARTE DIDÁCTICA

### Módulo: Bases de Datos — 1.º DAM

#### 2.1 Contextualización

- Aplicación directa en **desarrollo backend, administración de datos y consultas SQL**.
- Fundamenta operaciones en gestores como SQLite, MySQL o PostgreSQL.

#### 2.2 Objetivos de aprendizaje

- Comprender estructuras relacionales: tablas, atributos, claves.
- Aplicar operadores del álgebra relacional.
- Traducir álgebra  $\leftrightarrow$  SQL y razonar sobre optimización.
- Normalizar esquemas y detectar redundancia.

#### 2.3 Metodología

- Actividades prácticas en papel + SQL real.
- Ejercicios de traducción entre álgebra y SQL.
- Simulación de esquemas con DBDesigner, SQLite o db-fiddle.

#### 2.4 Atención a la diversidad

- Ejercicios escalonados: selección  $\rightarrow$  join  $\rightarrow$  subconsultas.
- Visualización de consultas con símbolos  $\pi$ ,  $\sigma$ ,  $\bowtie$ .
- Retos para alumnado avanzado: consultas con división y renombramiento.

#### 2.5 Principios DUA

- Representación: esquemas visuales, tablas, pseudocódigo.
- Acción: ejecución práctica en SGBD.
- Motivación: mostrar cómo SQL “piensa” como el álgebra relacional.

#### 2.6 Actividad principal

##### “Del papel a la base de datos: consultas en dos lenguajes”

1. Dado un esquema (clientes, pedidos), redactar consultas naturales.
2. Expresarlas en álgebra relacional formal.
3. Traducirlas a SQL y ejecutarlas.
4. Analizar con EXPLAIN y justificar decisiones.

#### 2.7 Evaluación

- Formativa + rúbrica por fases:
  - Precisión en álgebra.
  - Corrección sintáctica y lógica de SQL.
  - Justificación del diseño y comparación de eficiencia.

#### 2.8 Conclusión didáctica

- El álgebra relacional refuerza el pensamiento lógico y estructurado.
- Favorece la transición de “problema real”  $\rightarrow$  “consulta formal”  $\rightarrow$  “ejecución eficiente”.

## **Tema 72: Seguridad en sistemas de red: Servicios, protecciones, estándares avanzados**

### **1. Fundamentos de la seguridad en red**

#### **1.1. Importancia de la seguridad en red**

- Las redes son vectores de ataque constantes en entornos conectados.
- Riesgos: pérdida de disponibilidad, integridad y confidencialidad.

#### **1.2. Necesidad de protección**

- La ciberseguridad comienza en la red: proteger el canal de datos es esencial.

### **2. Servicios de seguridad**

#### **2.1. Autenticación y autorización**

- Métodos: contraseñas, autenticación multifactor (MFA), biometría.
- Protocolos: Kerberos, OAuth2, SSO (Single Sign-On).

#### **2.2. Control de acceso**

- Modelos: RBAC (control basado en roles), ABAC (basado en atributos).
- Tecnologías: VLANs, NAC (Network Access Control), listas de control de acceso (ACL).

#### **2.3. Cifrado y no repudio**

- Herramientas: HTTPS, VPN, cifrado de discos, firmas digitales.

#### **2.4. Auditoría y SIEM**

- SIEM: gestión centralizada de eventos e información de seguridad.
- Herramientas: Wazuh, Splunk; detección de anomalías mediante análisis en tiempo real.

### **3. Técnicas de protección**

#### **3.1. Segmentación de red**

- Uso de VLANs, microsegmentación y redes definidas por software (SDN).

#### **3.2. Bastionado (hardening)**

- Eliminación de servicios innecesarios, refuerzo de configuraciones, automatización con Ansible.

#### **3.3. Prevención de amenazas**

- Herramientas: EDR (Endpoint Detection and Response), DNSSEC, bloqueo de direcciones IP.

### **4. Defensa en profundidad**

#### **4.1. Firewalls de nueva generación (NGFW)**

- Inspección profunda de paquetes, filtrado por aplicación, bloqueo en tiempo real.

#### **4.2. Sistemas IDS e IPS**

- IDS: detección de intrusiones. IPS: prevención activa.

#### **4.3. Copias de seguridad**

- Regla 3-2-1: 3 copias, en 2 soportes diferentes, 1 externa.

### **5. Normativa y estándares**

#### **5.1. Estándares técnicos**

- ISO 27001, NIST SP 800-53, COBIT, MITRE ATT&CK.

#### **5.2. Legislación vigente**

- RGPD, LOPDGDD, ENS (Esquema Nacional de Seguridad), Directiva NIS2.

#### **5.3. Evaluación de riesgos**

- MAGERIT y PILAR: análisis detallado de activos, amenazas y vulnerabilidades.

### **6. Amenazas actuales**

- Man-in-the-Middle (MITM): interceptación si el tráfico no está cifrado.
- Ransomware: secuestro de datos mediante cifrado.
- Fallos de configuración en entornos cloud.
- Ataques DDoS: saturación de servicios mediante tráfico masivo.

### **7. Concienciación y formación**

- El usuario como primera línea de defensa.
- Simulacros y campañas de concienciación: phishing, ransomware, ingeniería social.
- Actividades de ciberseguridad: CyberCamp, CTFs, test de impacto.

## **PROPUESTA DIDÁCTICA: “CIBERDEFENSORES EN RED”**

### **A. Contextualización**

- Nivel educativo: 1.º FP Grado Superior en Administración de Sistemas Informáticos en Red (ASIR).
- Módulo: Seguridad y alta disponibilidad.

### **B. Objetivos de aprendizaje**

- Aplicar medidas de protección de red y respuesta a incidentes.
- Identificar amenazas y aplicar estándares y buenas prácticas.

- Fomentar la responsabilidad digital y el trabajo colaborativo.

### **C. Metodología**

- Aprendizaje basado en proyectos (ABP) y gamificación.
- Dinámica de roles: analista SIEM, responsable de red, backup, hardening.
- Aprendizaje activo mediante retos progresivos.

### **D. Actividad principal**

- Simulación de un SOC (Security Operations Center).
- Herramientas: TryHackMe, VirtualBox, Packet Tracer.
- Cada equipo diseña y defiende su infraestructura ante ataques simulados.
- Evaluación continua del rendimiento técnico y organizativo.

### **E. Atención a la diversidad (niveles III y IV)**

- Nivel III: apoyo visual, guías paso a paso, grupos heterogéneos.
- Nivel IV: adaptación de tareas, refuerzo individual, recursos accesibles.

### **F. Diseño Universal para el Aprendizaje (DUA)**

- Representación: vídeos, esquemas, simulaciones.
- Acción y expresión: elección de herramientas, roles diferenciados.
- Implicación: enfoque competitivo, trabajo por equipos, retroalimentación constante.

### **G. Evaluación**

- Rúbricas por competencias técnicas y actitudinales.
- Instrumentos: observación directa, diarios de aprendizaje, checklist de configuración.
- Criterios: efectividad defensiva, trabajo en equipo, resolución de incidentes.

### **H. Conclusión didáctica**

- La ciberseguridad es una competencia transversal y crítica.
- Simular un SOC permite integrar teoría, práctica y conciencia ética.
- El alumnado se convierte en protagonista de su aprendizaje, desarrollando competencias digitales avanzadas.

## **Tema 74: Sistemas multimedia**

### **1. Definición y contexto**

#### **1.1. ¿Qué es un sistema multimedia?**

- Conjunto de tecnologías que integran texto, imagen, audio, vídeo, animación y datos en tiempo real.
- Aplicaciones: educación (pizarras digitales), medicina (imagen diagnóstica), control remoto (drones), entretenimiento (videojuegos, RA).
- Incorporación de inteligencia artificial: reconocimiento facial, generación de voz e imagen.

### **2. Representación digital de medios**

#### **2.1. Imagen**

- Formatos RAW, BMP, sin compresión.
- Raster: JPEG, PNG (píxeles, pierden calidad al escalar).
- Vectorial: SVG (formas matemáticas, calidad escalable).
- Canal alfa: gestión de transparencia.

#### **2.2. Audio**

- Tasa de muestreo: frecuencia de captura (ej. 44.1 kHz).
- Bitrate: calidad versus tamaño.
- Formatos: WAV (sin compresión), FLAC (sin pérdida), MP3/AAC (con pérdida).

#### **2.3. Vídeo**

- FPS: fluidez (30 fps estándar, 60 fps mayor realismo).
- Códec: compresión (H.264); contenedor: empaquetado (MP4).

### **3. Procesamiento multimedia**

#### **3.1. Transformadas**

- Fourier: análisis de frecuencias (audio).
- DCT: base de JPEG.
- Wavelets: compresión multiescala (JPEG2000).

#### **3.2. Convoluciones**

- Aplicación de filtros a imágenes.
- Base de redes neuronales convolucionales (visión artificial).

### **4. Transmisión multimedia**

- Protocolos adaptativos: HLS, DASH (ajuste de calidad).
- Protocolos en tiempo real: RTMP, RTSP (baja latencia).

### **5. Inteligencia Artificial en multimedia**

#### **5.1. Generación**

- DALL·E, Stable Diffusion, voice cloning, NeRF.

#### **5.2. Análisis**

- YOLO, DETR (detección objetos).
- CLIP, GPT-4V (relación imagen-texto).

### **6. Herramientas**

- FFmpeg: conversión, edición por línea de comandos.
- OpenCV: visión artificial.
- MediaPipe: detección de gestos en móviles.

### **7. Tendencias futuras**

- Codificación neural, vídeo volumétrico, edge computing, interfaces adaptativas.

### **8. Ética y legislación**

- Deepfakes y manipulación audiovisual.
- Sesgos algorítmicos.
- IA Act (UE): marco legal según nivel de riesgo.

### **9. Conclusión**

- Los sistemas multimedia evolucionan hacia la comprensión y generación inteligentes de contenido.
- Su diseño debe equilibrar eficiencia técnica, ética y usabilidad.

## **PROPUESTA DIDÁCTICA: “ENTRENADOR MULTIMEDIA: CREA UNA APP DE FITNESS INTERACTIVO”**

### **A. Contextualización**

- Nivel educativo: 2.º curso de Grado Superior en DAM.
- Módulo: Multimedia y dispositivos móviles.

### **B. Objetivos de aprendizaje**

- Integrar medios audiovisuales en apps Android.

- Optimizar la compresión y la reproducción multimedia.
- Diseñar experiencias interactivas y accesibles.

### **C. Metodología**

- Aprendizaje basado en proyectos (ABP).
- Trabajo en equipo con división de roles técnicos.

### **D. Actividad principal**

- Proyecto: APP Android de entrenador personal.
- Funcionalidades:
  - Vídeos de ejercicios grabados o de libre uso.
  - Audios de instrucciones y motivación.
  - Temporizador configurable para rutinas.
  - Dinamizador virtual con mensajes automáticos.
  - Modo offline y control de resolución/bitrate.
- Herramientas: Android Studio, FFmpeg, CapCut, OBS Studio.

### **E. Atención a la diversidad**

- Nivel III: plantillas, videotutoriales, apoyo técnico continuo.
- Nivel IV: descomposición de tareas, soporte individualizado.

### **F. DUA**

- Representación: vídeos subtítulos, interfaces intuitivas.
- Expresión: variedad de herramientas y temas.
- Implicación: aplicación real, presentación gamificada.

### **G. Evaluación**

- Rúbricas: integración técnica, diseño, accesibilidad y documentación.
- Instrumentos: presentación oral, prueba funcional, memoria técnica.

### **H. Conclusión didáctica**

- Desarrollo de competencias en programación, tratamiento multimedia y ética digital.
- La app como producto funcional, motivador y aplicable en contextos reales.