Oposiciones cuerpo de secundaria.

Esquemas sobre temario oposición profesorado Secundaria.

Especialidad informática

Autor: Sergi García Barea

Actualizado Abril 2025

Licencia

Índice

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 **tres 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 éste aplicado a un determinado nivel previamente establecido por la persona aspirante. Finalizada la exposición, el tribunal podrá realizar un debate con la persona candidata sobre el contenido de su intervención."

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

Tema 1: Representación y comunicación de la información

1. Introducción

- Representación de la información fundamental
- Influye en como se almacenan, procesan y transmiten datos.
    - Una correcta elección lo hace eficiente.
- Se relaciona con el procedimiento eficiente

2. Sistemas de numeración

Conceptos básicos

- **Base**: Número de símbolos utilizados en un sistema de numeración.
- **Dígitos**: Símbolos permitidos en cada base.
- **Representación posicional**: Valor de un dígito depende de su posición en el número.

Principales sistemas

- **Decimal (base 10)**: Usado en la vida cotidiana.
- **Binario (base 2)**: Fundamental en computación.
- **Octal (base 8)** y **Hexadecimal (base 16)**: común en computación por su fácil paso de binario a la base y viceversa (3 bits -> Octal, 4 bits -> Hexadecimal).

3. Conversiones entre bases

Conversión entre bases (decimal-binario, binario-hexadecimal, etc.) con aplicaciones en informática, programación y hardware.

4. Sistema binario

- **Bits y bytes**: Unidad básica de información (8 bits = 1 byte).
- **Representación de datos**:
    - Números: Enteros y en punto flotante.
    - Caracteres: Códigos ASCII y Unicode.
    - Imágenes y sonido: Codificación binaria en formatos digitales.

5. Operaciones en binario

- **Aritmética binaria**: Suma, resta, multiplicación y división.
- **Representación de números**:
    - Enteros sin signo y con signo (complemento a 1 y 2).
    - Punto flotante (IEEE 754).
- **Operaciones lógicas**: AND, OR, XOR, NOT.

6. Sistemas octal y hexadecimal

- **Relación con el binario**: Facilitan la representación compacta de datos (3 bits octal, 4 bits binario)
- **Aplicaciones**:
    - Direcciones de memoria.
    - Depuración a bajo nivel.
    - Trucos en videojuegos.
    - Programación en ensamblador.
    - Visualización de hash.

7. Códigos binarios

7.1 Códigos numéricos

- **BCD (Binary-Coded Decimal)**: Representación decimal en binario.
- **Exceso-3**: Variante de BCD.
- **Código Gray**: Usado en conversión A/D y circuitos digitales.

7.2 Códigos alfanuméricos

- **ASCII (7 y 8 bits)**: Representación de caracteres.
- **Unicode (UTF-8, UTF-16, UTF-32)**: Soporte para idiomas y caracteres especiales.

8. Redundancia y detección de errores

- **Necesidad**: Corrección en transmisiones digitales.
- **Códigos de detección de errores**:
    - Paridad.
    - CRC (Cyclic Redundancy Check).
- **Códigos de corrección de errores**:
    - Hamming.
    - Reed-Solomon (CDs, DVDs, QR, almacenamiento).

9. Seguridad informática y representación de la información

9.1 Funciones hash

- **Propiedades**: Determinismo, resistencia a colisiones, difusión.
- **Algoritmos comunes**:
    - MD5 (obsoleto).
    - SHA-1 (obsoleto).
    - SHA-256 (seguro, usado en criptografía y blockchain).
- **Aplicaciones**:
    - Integridad de datos.
    - Firmas digitales y autenticación.
    - Almacenamiento seguro de contraseñas (hash + salts).

9.2 Ataques y defensa en codificación de información

    - **Tablas rainbow**: Precomputación de hashes de contraseñas.
        - **Mitigación**: Salts, bcrypt, PBKDF2, Argon2.
    - **Criptoanálisis**:
        - Ataques de colisión en funciones hash.
        - Ataques de fuerza bruta.
    - **Cifrado en comunicación**:
        - Algoritmos simétricos (AES).
        - Algoritmos asimétricos (RSA, ECC).
        - Uso en HTTPS, VPN y almacenamiento seguro.

10. Comunicación de la información

10.1 Elementos de un sistema de comunicación digital

- **Modelo de Shannon y Weaver**.
- **Tipos de señales**: Analógicas y digitales.
- **Medios de transmisión**: Cableados (fibra óptica, cobre) e inalámbricos (Wi-Fi, Bluetooth).
- **Protocolos y estándares**: TCP/IP, Ethernet, Wi-Fi.

10.2 Compresión de datos

- **Compresión sin pérdida**:
    - Huffman.
    - Lempel-Ziv (LZ77, LZ78, LZW).
- **Compresión con pérdida**:
    - JPEG (imágenes).
    - MP3 (audio).
    - H.264 (video).
- **Aplicaciones**: Transmisión y almacenamiento eficiente de datos.

Tema 2: Elementos funcionales de un ordenador digital. Arquitectura

1. Introducción

1.1. Modelo de Von Neumann
- Arquitectura con una única memoria compartida para datos e instrucciones.
- Ejecución secuencial de instrucciones.
- Problema del "cuello de botella de Von Neumann": la velocidad del bus de memoria limita el rendimiento.

1.2. Arquitectura Harvard (comparación con Von Neumann)
- Memoria separada para datos e instrucciones.
- Mayor eficiencia en acceso simultáneo a memoria.
- Uso en DSPs y microcontroladores modernos.

2. Memoria principal

2.1. Jerarquía de memoria
- Criterios: tiempo de acceso, capacidad, coste.
- Estructura moderna:
    - Registros (CPU, rápidos y pequeños)
    - Caché (L1, L2, L3) (cercana a la CPU, rápida)
    - RAM (almacenamiento temporal principal)
    - SSD/HDD (almacenamiento secundario)
    - Almacenamiento en red/Nube (remoto y escalable)
- Tendencias:
    - Memorias no volátiles rápidas (Optane, Z-NAND).
    - Integración de memoria en el chip (SoC, Apple M-series, AMD 3D V-Cache).

2.2. Características clave
- Direccionamiento: 32-bit, 64-bit.
- Longitud de palabra: impacta en ancho de banda y rendimiento.
- Operaciones básicas: lectura (Read) y escritura (Write).
- Parámetros:
    - Tiempo de acceso (latencia).
    - Tiempo de ciclo (frecuencia de acceso).

2.3. Tipos de memoria
- RAM (Random Access Memory):
    - DRAM (Dynamic RAM): DDR4, DDR5.
    - SRAM (Static RAM): utilizada en caché.
- ROM (Read-Only Memory):
    - PROM, EPROM, EEPROM.
- Memoria flash: usada en SSD, BIOS/UEFI.

2.4. Optimización de memoria
- Memoria caché: niveles L1, L2, L3, optimización con algoritmos de reemplazo.
- Memoria virtual: paginación, segmentación, swap.
- Tendencias: Memoria unificada en GPUs modernas (HBM, VRAM GDDR6).

3. Unidad Central de Proceso (CPU)

3.1. Registros
- Registros de propósito general (GP).
- Registros del sistema:
    - ALU, Contador de Programa (PC), Registro de Instrucción (IR).
    - Registros de estado (flags).
    - Registros de memoria (MAR, MDR).

3.2. Unidad Aritmético-Lógica (ALU)
- Realiza operaciones matemáticas y lógicas.
- Soporte moderno:
    - SIMD (Single Instruction Multiple Data) – AVX, SSE.
    - FPU (procesamiento en coma flotante).

3.3. Unidad de Control
- Funciones:
    - Gestiona la ejecución de instrucciones.
    - Genera señales de control.
- Implementación:
    - Cableada (hardware dedicado).
    - Microprogramada (almacena microinstrucciones).

4. Unidad de Entrada/Salida (E/S)

4.1. Direccionamiento
- Mapa de memoria único: unifica memoria y E/S.
- Mapa de memoria independiente: memoria y E/S separadas.

4.2. Modos de transferencia
- Controlada por programa: ineficiente.
- Interrupciones: el hardware notifica a la CPU.
- DMA (Direct Memory Access): acceso directo a memoria sin CPU.

4.3. Transmisión
- Síncrona: reloj compartido.
- Asíncrona: con señales de inicio/parada.
- Interfaces modernas:
    - USB 4, Thunderbolt 4, PCIe 5.0, NVMe.

5. Buses

5.1. Tipos de buses
- Bus de datos: transporta información.
- Bus de direcciones: identifica posiciones de memoria.
- Bus de control: coordina señales.

5.2. Clasificación
- Internos: dentro de la CPU.
- Externos: conectan con dispositivos.

5.3. Temporización
- Síncrona: reloj común.
- Asíncrona: sin sincronización estricta.
- Ejemplos modernos:
    - PCIe 5.0 (gráficas, SSD).
    - NVMe (almacenamiento rápido).
    - USB 4, Thunderbolt.

6. Ciclo de instrucción

6.1. Formato de la instrucción
- Estructura: código de operación (opcode) + operandos.
- Tipos:
    - Longitud fija (RISC).
    - Longitud variable (CISC).

6.2. Fases del ciclo de instrucción
- Fetch: búsqueda de la instrucción.
- Decode: decodificación.
- Execute: ejecución.
- Memory Access: acceso a memoria (si aplica).
- Write Back: almacenamiento de resultados.

6.3. Optimizaciones modernas
- Pipeline: ejecución en paralelo de fases.
- Superescalaridad: ejecución de varias instrucciones simultáneas.
- Ejecución fuera de orden (OoO): mejora el rendimiento en CPUs modernas.
- Predicción de saltos: evita penalizaciones por bifurcaciones.
- Procesamiento paralelo:
    - Multi-core, Hyper-Threading.
    - Computación en GPU para tareas paralelizables.

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

1. Introducción

La CPU es el "cerebro" del ordenador, encargada de procesar instrucciones y coordinar operaciones.

Evolución:

- Mononúcleo → Primeras CPUs (8086, Pentium).
- Multinúcleo → Mejora rendimiento (Core 2 Duo, Ryzen).
- Arquitecturas híbridas → Núcleos de alto rendimiento y eficiencia (Intel Alder Lake, ARM big.LITTLE).

Tendencias actuales:

- IA en CPU: Instrucciones especializadas (Intel DL Boost, Apple Neural Engine).
- Arquitecturas heterogéneas: Integración CPU-GPU en chips como Apple M1/M2 y AMD Ryzen con gráficos integrados.
- La tendencia actual es optimizar eficiencia energética sin perder rendimiento. Esto es clave en dispositivos móviles y servidores.

2. Estructura de la CPU

2.1. Unidad Aritmético-Lógica (ALU)

- Ejecuta operaciones matemáticas y lógicas.
    - Registros internos: Acumulador, operandos, flags (estado del cálculo)
    - Soporta cálculos avanzados con unidades especializadas:
        - FPU (Floating Point Unit) → Cálculo en coma flotante.
        - SIMD, AVX, SSE → Procesamiento vectorial en paralelo (importante en gráficos y ciencia de datos).
            - La CPU actuales incluyen unidades vectoriales (AVX-512, ARM NEON) para mejorar rendimiento en IA y multimedia.

2.2. Unidad de Control (UC)

- Coordina la ejecución de instrucciones:
    - Decodifica instrucciones y genera señales de control.
    - Regula el flujo de datos entre ALU, registros y memoria.
- Implementaciones:
    - Cableada → Rápida pero inflexible (usada en CPUs sencillas).
    - Microprogramada → Más flexible y fácil de actualizar (usada en procesadores modernos).
- Técnicas modernas de optimización:
    - Pipelining → Se solapan varias instrucciones a la vez.
    - Ejecución especulativa → Se anticipan operaciones antes de confirmarlas.

    - Paralelismo a nivel de instrucción (ILP) → Se ejecutan múltiples instrucciones simultáneamente.
    - TENDENCIA: Predicción de saltos con IA para reducir tiempos muertos en ejecución.

2.3. Memoria Interna

2.3.1. Registros
- Memoria ultrarrápida dentro de la CPU.
- Tipos clave:
    - Generales → Datos temporales.
    - Especiales → PC (contador de programa), IR (registro de instrucción), FLAGS (estado), MAR/MDR (acceso a memoria).

2.3.2. Memoria Caché
- Minimiza la latencia al reducir accesos a RAM.
- Jerarquía:
    - L1 → Más rápida, menor tamaño.
    - L2 → Mayor capacidad, compartida por varios núcleos.
    - L3 → Accesible por toda la CPU, pero más lenta.
- Técnicas clave:
    - Prefetching → Anticipa datos antes de ser necesarios.
    - Coherencia de caché → Evita inconsistencias en sistemas multinúcleo.
    - TENDENCIA: Caché más inteligente y adaptativa (Intel Adaptive Boost en CPUs recientes).

2.3.3. Memoria RAM
- Espacio de trabajo de la CPU.
- Tipos actuales: DDR5, LPDDR5X (móviles y portátiles de bajo consumo).

2.4. Buses Internos
- Bus de Datos → Transporta información.
- Bus de Direcciones → Localiza datos en memoria.
- Bus de Control → Coordina el tráfico de datos.
- Evolución:
    - FSB (Front-Side Bus) → Obsoleto.
    - QPI (Intel), Infinity Fabric (AMD), NVLink (NVIDIA) → Mejor comunicación interna.
    - TENDENCIA: Comunicación de altísima velocidad con memoria y GPU (AMD 3D V-Cache, Apple Unified Memory).

3. Funcionamiento de la CPU

3.1. Instrucciones
- Conjunto de instrucciones:
    - CISC (x86, ARMv8-A en móviles) → Más instrucciones, más complejas.
    - RISC (ARM, RISC-V, Apple M1/M2, PowerPC) → Instrucciones más simples, más rápidas.

- Extensiones modernas:
  - AVX-512 → Acelera cálculos científicos y multimedia.
  - Intel VT-x / AMD-V → Soporte para virtualización.
  - TENDENCIA: Crecimiento de RISC-V, una arquitectura abierta que puede reemplazar x86 y ARM en muchos sectores.

3.2. Ciclo de Instrucción

- Fases:
  1️⃣ Fetch → Busca la instrucción en memoria.
  2️⃣ Decode → Decodifica y determina qué hacer.
  3️⃣ Execute → ALU/FPU realiza la operación.
  4️⃣ Memory Access → Si es necesario, accede a memoria.
  5️⃣ Write-back → Guarda el resultado.
- Optimización moderna:
  - Ejecución fuera de orden (OoO) → Ejecuta instrucciones en el orden más eficiente.
  - Predicción de saltos → Minimiza esperas en bifurcaciones de código.
  - Hyper-Threading (Intel) / SMT (AMD) → Simula varios hilos por núcleo para mejorar rendimiento.
  - TENDENCIA: Procesamiento de IA directamente en CPU (Intel AMX, Apple Neural Engine).

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

1. INTRODUCCIÓN

- Importancia de la memoria en la arquitectura del computador.
- Impacto en el rendimiento: Cuellos de botella en acceso a datos.

2. MEMORIA: CONCEPTOS FUNDAMENTALES

2.1. Elementos clave de una memoria

- Soporte físico: Silicio (RAM, Flash), magnético (HDD), óptico (CD/DVD).
- Acceso: Aleatorio (RAM, SSD), secuencial (cintas), asociativo (caché).
- Durabilidad: Volátil (RAM) vs. No volátil (Flash, HDD).

2.2. Direccionamiento

- Bidimensional (2D): Un solo decodificador (memorias pequeñas).
- Tridimensional (3D): Uso de múltiples decodificadores para grandes volúmenes de datos.

2.3. Características clave

- Velocidad: Latencia y ancho de banda.
- Unidad de transferencia: Palabra, bloque, línea de caché.
- Modos de direccionamiento: Directo, indirecto, segmentado, paginado.

3. TIPOS DE MEMORIAS

3.1. Volátiles (rápidas, almacenamiento temporal)

- SRAM: Usa biestables, rápida, cara, caché de CPU.
- DRAM: Usa condensadores, requiere refresco, más lenta pero más densa.
- SDRAM: Sincronizada con el bus de memoria, mejora eficiencia.
- DDR (DDR1-5): Aumenta frecuencia y reduce consumo con cada versión.
- GDDR (GDDR5-6X): Optimizada para GPU (mayor ancho de banda).
- HBM: Apilamiento vertical, alto ancho de banda (IA, servidores).

3.2. No volátiles (almacenamiento permanente)

- ROM: Solo lectura, firmware.
- Flash (NAND/NOR): Base de SSD y almacenamiento portátil.
- NVRAM: Combinación de velocidad RAM y persistencia ROM.

4. JERARQUÍA DE MEMORIAS Y FUNCIONES

- Registros: Dentro de la CPU, acceso instantáneo.
- Caché (L1, L2, L3): Reduce latencia con RAM.
- RAM: Almacena datos en ejecución.
- Almacenamiento secundario (SSD, HDD): Datos permanentes.

5. MEMORIA PRINCIPAL Y CONEXIÓN CON CPU

5.1. Estructura
- SRAM: Matriz bidimensional, más rápida.
- DRAM: Celdas con condensadores, requiere refresco.
- ROM: Direccionamiento fijo.

5.2. Dirección y acceso
- Buses: Direcciones, datos, control.
- Modos de acceso: Lectura-Modificación-Escritura, paginación, acceso por columna.
- Refresco DRAM: Distribuido o en ráfagas.

6. MEJORAS DE RENDIMIENTO

6.1. Memoria Caché
- Tipos: L1 (ultrarrápida), L2 (intermedia), L3 (compartida en CPU).
- Mapeos: Directo, asociativo, por conjuntos.
- Reemplazo: LRU, FIFO, Aleatorio.

6.2. Memoria Virtual
- Extiende la RAM usando almacenamiento secundario.
- Traducción de direcciones: MMU convierte direcciones virtuales en físicas.
- Técnicas:
    - Paginación: División fija en páginas, usa tabla de páginas.
    - Segmentación: División lógica del programa.
    - Segmentación paginada: Híbrido, más flexible.
- TLB (Translation Lookaside Buffer): Acelera la traducción.

7. TECNOLOGÍAS MODERNAS
- Memorias persistentes (PMEM)
    - Ejemplo: Intel Optane.
    - Ventaja: Actúa como RAM pero retiene datos tras apagado.
- Memoria apilada (3D XPoint)
    - Densidad superior y baja latencia.
    - Mejor que NAND en acceso aleatorio.
- Memoria computacional (Processing In Memory, PIM)
    - Procesamiento dentro de la memoria (menos movimiento de datos → más rendimiento).
    - Usada en IA y HPC.

Tema 10: Representación Interna de los Datos

1. Introducción
- Los ordenadores representan internamente los datos en binario (base 2).
- Se utilizan diferentes bases según el contexto:
    - Binario (base 2): Electrónica, instrucciones de máquina.
    - Octal (base 8): Representación compacta en sistemas antiguos.
    - Decimal (base 10): Interacción con humanos.
    - Hexadecimal (base 16): Representación eficiente de direcciones y colores.
- Importancia de los cambios de base en informática y redes.

2. Representación de Caracteres Alfanuméricos
- ASCII (7/8 bits): Estándar básico, solo caracteres en inglés.
- EBCDIC: Formato IBM, en desuso.
- UNICODE (UTF-8, UTF-16, UTF-32): Soporta todos los idiomas, emojis y símbolos matemáticos.
    - Nota: UTF-8 es el más usado en la web, ya que es eficiente en espacio.

3. Representación de Datos Booleanos
- 0 → Falso, 1 → Verdadero.
- Uso en circuitos lógicos: Álgebra de Boole, puertas lógicas.
- Simplificación con mapas de Karnaugh: Optimización de hardware.

4. Representación de Datos Enteros
- Signo magnitud: Representación básica pero con el problema del doble 0.
- Complemento a 1 (CA1): Mejor que el anterior, pero aún con el problema del doble 0.
- Complemento a 2 (CA2):
    - Usado en procesadores modernos.
    - Permite representar negativos sin ambigüedades.
    - Facilita operaciones aritméticas (restar es sumar el negativo).
    - Ejemplo: En 8 bits, -1 es 11111111 y 1 es 00000001.
- Exceso Z: Se usa en exponentes de coma flotante para evitar negativos.

5. Representación de Datos Reales
- Coma fija: Obsoleta por su baja precisión.
- Coma flotante (IEEE 754): Estándar actual.
    - 32 bits (simple precisión): Uso general.
    - 64 bits (doble precisión): Cálculo científico.
    - 128 bits (cuádruple precisión): Supercomputación.
- Conceptos clave:
    - Normalización: Mantiene la representación eficiente.
    - Desbordamiento/subdesbordamiento: Problemas con valores fuera de rango.

6. Representación de Números Complejos
- Se almacena como parte real + parte imaginaria en coma flotante.

- Uso en:
    - Procesamiento de señales: Audio, radio, telecomunicaciones.
    - Computación cuántica: Algoritmos avanzados.
    - Gráficos 3D: Motores de videojuegos y diseño CAD.

7. Representación Interna de Estructuras de Datos

- Vectores y matrices: Acceso secuencial o por filas/columnas.
- Listas enlazadas: Eficientes para inserciones/borrados dinámicos.
- Árboles:
    - BST (Árbol de Búsqueda Binaria): Estructura básica.
    - AVL, B+: Equilibrados para optimizar búsquedas.
- Grafos:
    - Representados con listas de adyacencia (eficientes en espacio) o matrices (búsqueda rápida).
    - Uso en redes, inteligencia artificial y rutas GPS.
- Tablas hash:
    - Búsquedas en tiempo constante O(1).
    - Usadas en bases de datos, almacenamiento en caché.
- Punteros y estructuras dinámicas: Base de lenguajes como C/C++.

8. Representación interna de elementos multimedia

- Gráficos vectoriales: Formatos SVG, PDF (se pueden escalar sin perder calidad).
- Mapas de bits (raster): Formatos BMP, PNG, JPEG, WebP.
- Compresión de imágenes:
    - Con pérdida: JPEG, HEIC → Ahorra espacio sacrificando calidad.
    - Sin pérdida: PNG, FLAC → Mantiene fidelidad.
- Gráficos 3D: Modelos OBJ, FBX, GLTF usados en videojuegos y realidad virtual.
- Sonido: distintos codecs WAV, MP3, OGG
- Video: distintos codecs MPEG, MP4, etc.

9. Cifrado y Compresión

- Cifrado moderno:
    - AES: Algoritmo estándar de cifrado simétrico.
    - RSA: Cifrado asimétrico usado en comunicaciones seguras.
    - ECC (Curvas Elípticas): Más eficiente que RSA con claves más cortas.
    - Criptografía post-cuántica: En desarrollo para resistir ataques de ordenadores cuánticos.
- Compresión:
    - Sin pérdida: ZIP, PNG, FLAC → Se pueden recuperar los datos exactos.
    - Con pérdida: MP3, JPEG, H.265 → Reduce tamaño eliminando información irrelevante.
- Funciones de resumen (Hash)

Tema 11: Organización Lógica de los Datos. Estructuras Estáticas

1. Introducción
- Abstracción de datos: Separación entre representación lógica y física.
- Tipos de datos: Definidos como un conjunto de valores con operaciones asociadas.
- Importancia: Fundamental en el diseño de algoritmos eficientes.

2. Tipos Abstractos de Datos (TAD)
- Definición: Modelos estructurados independientes de la implementación.
- Componentes:
    - Datos.
    - Operaciones.
    - Propiedades semánticas.
- Ejemplos de TADs:
    - Pila (Stack): Estructura LIFO (Last In, First Out).
    - Cola (Queue): Estructura FIFO (First In, First Out).
    - Lista: Secuencia ordenada de elementos.
    - Árbol: Estructura jerárquica con nodos y relaciones padre-hijo.
    - Grafo: Modelado de relaciones en redes y sistemas complejos.

3. Tipos Escalares

3.1 Tipos Normalizados
- Entero: Representado en complemento a 2.
- Real: IEEE 754, con formatos de distinta precisión.
- Carácter: Codificado en Unicode (UTF-8, UTF-16).
- Booleano: Representa valores lógicos (0/1).

3.2 Tipos Escalares Definidos por el Usuario
- Enumeración: Conjunto de valores predefinidos (Ejemplo: {Rojo, Verde, Azul}).
- Rango: Subconjunto de un tipo base (Ejemplo: 1..100).

4. Tipos Estructurados
- Definición: Agrupación de múltiples valores en una entidad.

4.1 Vectores (Arrays)
- Unidimensional: Manejo de cadenas de caracteres.
- Multidimensional: Representación de matrices e imágenes.

4.2 Conjuntos
- Operaciones matemáticas eficientes: Unión, intersección, diferencia.

4.3 Registros (Structs, Tuplas)
- Normales: Contienen varios tipos de datos (Ejemplo: nombre: String, edad: Int).
- Variantes: Combinan parte fija y parte variable (Ejemplo: union en C).

5. Implementación Estática de Estructuras de Datos

5.1 Pilas (Stacks)

- Implementación con arrays y puntero al tope.
- Operaciones básicas: push(), pop(), top().
- Usos: Evaluación de expresiones, recursión, control de llamadas a funciones.

5.2 Colas (Queues y Deques)

- Implementadas con arrays circulares.
- Operaciones: enqueue(), dequeue(), front().
- Usos: Planificación de procesos, manejo de tareas en redes.

5.3 Listas

- Listas enlazadas vs. Arrays estáticos.
- Ventajas de arrays: Acceso rápido a elementos.
- Desventajas: Inserciones y borrados costosos.

5.4 Árboles

- Ejemplo: Árboles binarios de búsqueda (BST), AVL.
- Implementación en arrays: Hijo izquierdo en 2i+1, hijo derecho en 2i+2.
- Usos: Bases de datos, compiladores, organización jerárquica de información.

5.5 Grafos

- Representación:
    - Matriz de adyacencia: Más memoria, acceso inmediato.
    - Lista de adyacencia: Más eficiente en grafos dispersos.
- Usos: Redes, IA, algoritmos de caminos más cortos (Dijkstra, A*).

5.6 Funciones y tablas Hash (resumen)

- Uso de arrays y colisiones manejadas con encadenamiento o direccionamiento abierto.
- Usos: Bases de datos, autenticación, almacenamiento en caché.

6. Concursos de Programación

Los concursos de programación fomentan el pensamiento algorítmico y el dominio de estructuras de datos.

6.1 Olimpiada Informática Española (OIE)

- Competencia nacional para estudiantes preuniversitarios.
- Fases: Regionales, nacionales y clasificación a la Olimpiada Informática Internacional (IOI).
- Desafíos: Algoritmos avanzados, estructuras de datos eficientes.

6.2 ProgramaMe

- Concurso de programación para estudiantes de FP.
- Modalidad por equipos.
- Uso de estructuras de datos estáticas y dinámicas.
- Pruebas de eficiencia y optimización de código.

6.3 Importancia en la Enseñanza

- Desarrollo de habilidades algorítmicas.

- Aplicación práctica de estructuras de datos.
- Entrenamiento en plataformas como Codeforces, LeetCode, AtCoder.

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

1. Introducción

- Definición de sistema operativo (SO).
- Funciones principales: gestión de hardware, procesos, memoria, usuarios y redes.
- Evolución de los sistemas operativos: desde monousuario a sistemas distribuidos y en la nube.

2. Clasificación de los sistemas operativos

2.1. Según el número de procesadores

- Sistemas monoprocesador: Ejecutan instrucciones en un solo núcleo.
- Sistemas multiprocesador: Permiten ejecución en varias CPU (SMP, NUMA).

2.2. Según el número de usuarios

- Sistemas monousuario: Un solo usuario a la vez (Ejemplo: Windows 11, macOS Sonoma).
- Sistemas multiusuario: Varios usuarios simultáneos (Ejemplo: Linux Ubuntu Server, Windows Server 2022).

2.3. Según el número de tareas

- Monotarea: Ejecutan una tarea a la vez (obsoletos).
- Multitarea: Ejecutan múltiples procesos en paralelo (Windows, Linux, macOS).

2.4. Según la arquitectura del núcleo

- Monolítico: Todo el SO opera en modo núcleo (Linux tradicional).
- Microkernel: Mínimos servicios en el núcleo, resto en espacio de usuario (Minix, QNX).
- Híbrido: Combinación de ambos (Windows NT, macOS).

2.5. Según el entorno de ejecución

- Sistemas operativos en red: Soportan comunicación y compartición de recursos (Windows Server, FreeBSD).
- Sistemas operativos distribuidos: Varias máquinas actúan como un solo sistema (Kubernetes, Apache Mesos).
- Sistemas operativos en la nube: Desplegados en infraestructura cloud (Chrome OS, AWS Lambda, Azure Sphere).

2.6. Según el tiempo de respuesta

- Tiempo real (RTOS): Respuesta garantizada en tiempo crítico (VxWorks, QNX).

2.7. Sistemas operativos emergentes

- IoT: Adaptados a dispositivos de bajo consumo (Zephyr, RIOT OS).
- Computación cuántica: Coordinan acceso a qubits (Qiskit, Cirq).

3. Explotación de sistemas operativos monousuario

3.1. Procedimientos de explotación

- Instalación y configuración inicial.
- Gestión de usuarios y permisos.
- Administración de software y actualizaciones.

3.2. Niveles de explotación

- Usuario: Operaciones básicas (configuración, ejecución de aplicaciones).
- Administrador: Gestión avanzada del sistema (usuarios, recursos, copia, seguridad)

3.3. Ejemplo de sistema monousuario moderno

- Windows 11: Configuración de usuarios, administración de políticas de seguridad, actualizaciones automáticas con Windows Update.
- macOS Sonoma: Gestión de perfiles de usuario, optimización de recursos, integración con iCloud y seguridad avanzada.

4. Administración de sistemas operativos multiusuario

4.1. Multitarea y gestión de procesos

- Planificadores de procesos: FIFO, Round Robin, Prioridades.
- Comunicación entre procesos (IPC, sockets, colas de mensajes).

4.2. Gestión de memoria

- Memoria virtual: paginación, segmentación, swapping.
- Espacios de direcciones de usuario y kernel.

4.3. Servicios en sistemas multiusuario

4.3.1. Servicios en UNIX/Linux

- Daemons, systemd, cron jobs.

4.3.2. Servicios en Windows

- Servicios del sistema, Windows Task Scheduler.

4.4. Gestión de almacenamiento

- Sistemas de archivos modernos: NTFS, EXT4, Btrfs, ZFS.
- Gestión de volúmenes lógicos (LVM).

4.5. Administración en sistemas multiusuario modernos

- Linux Ubuntu Server: Gestión de permisos con sudo, ACLs, control de recursos con cgroups.
- Windows Server 2022: Administración de usuarios y grupos, Active Directory, gestión de políticas con GPO.

5. Virtualización y contenedores

5.1. Tipos de virtualización

- Virtualización completa: Virtualiza hardware completamente (VMware ESXi, VirtualBox, Hyper-V).
- Paravirtualización: Virtualiza parcialmente (Xen, KVM).
- Virtualización a nivel de sistema operativo: Contenedores (Docker, LXC).

5.2. Contenedores y orquestación

- Docker: Creación y administración de contenedores.
- Podman: Alternativa a Docker sin daemon centralizado.
- Kubernetes: Orquestación de contenedores a gran escala.

5.3. Comparación entre máquinas virtuales y contenedores

Característica	**Máquinas Virtuales (VMs)**	**Contenedores**
Aislamiento	Alto (SO independiente)	Medio (mismo kernel)
Rendimiento	Menor (virtualización completa)	Mayor (comparten kernel)
Uso de recursos	Alto	Bajo

Tiempo de arranque	Lento	Rápido

6. Seguridad y administración avanzada

6.1. Seguridad en sistemas operativos

- Control de acceso: autenticación multifactor, permisos, ACLs.
- Seguridad en archivos y discos: BitLocker, LUKS, cifrado ZFS.
- Seguridad en redes: firewalls (iptables, Windows Defender Firewall).

6.2. Monitorización y rendimiento

- Linux: htop, atop, iostat, Prometheus.
- Windows: Monitor de rendimiento, Sysinternals.

6.3. Administración en la nube y Edge Computing

- Infraestructura como servicio (IaaS): AWS EC2, Azure VMs, Google Compute Engine.
- Plataforma como servicio (PaaS): AWS Lambda, Azure Functions.
- Edge Computing: Procesamiento local en dispositivos para baja latencia (AWS Greengrass, Azure IoT Edge).

7. Tendencias y futuro de los sistemas operativos

- Sistemas operativos con IA: Adaptación dinámica a uso y seguridad (Windows Copilot, macOS AI Features).
- Computación cuántica: SO optimizados para procesadores cuánticos (IBM Q, Microsoft Quantum).
- Serverless OS: Eliminación de administración de servidores (AWS Lambda, Cloud Run).

8. Conclusión

- Evolución hacia SO altamente optimizados para virtualización, contenedores y cloud computing.
- Importancia de la seguridad y monitorización avanzada.
- Desafíos futuros en escalabilidad, seguridad y eficiencia energética.

Tema 21: Sistemas informáticos. Estructura física y funcional

1. Introducción

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

2. Sistemas Informáticos

2.1. Definición y Evolución

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

2.2. Clasificación

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

2.3. Tendencias Actuales

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

3. Estructura Física y Funcional

3.1. Hardware

3.1.1. Arquitectura Von Neumann:

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

3.1.2. CPU:

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

3.1.3. Memoria y Almacenamiento:

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

3.1.4. Periféricos:

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

3.1.5. Redes:

- SDN, Wi-Fi 6, Edge Computing.

3.2. Software

3.2.1. Programación:

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

3.2.2. Aplicación:

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

3.2.3. Sistema:

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

3.2.4. Virtualización:

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

3.2.5. Seguridad:

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

4. Conclusiones

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

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

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

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

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

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

5. Instalación y Configuración
- 5.1. Ubicación física:

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

6. Gestión de Configuración

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

7. Seguridad en Sistemas

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

8. Procedimientos de Uso

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

9. Conclusiones

- Tendencias clave:
    - Automatización (DevOps/GitOps), seguridad Zero Trust, hybrid cloud.
- Retos futuros:

- Ciberseguridad, eficiencia energética en Data Centers.

Tema 23: Diseño de algoritmos. Técnicas descriptivas.

1. **Introducción**

1.1 Concepto de algoritmo

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

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

1.2 Importancia de los algoritmos en computación

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

1.3 Evolución histórica

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

2. Elementos básicos de los algoritmos

2.1 Acciones fundamentales

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

2.2 Estructuras de control

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

3. Representación de algoritmos

3.1 Pseudocódigo

Ejemplo (factorial):

```
INICIO
   LEER(n)
```

```
  factorial = 1
  PARA i DESDE 1 HASTA n HACER
    factorial = factorial * i
  FIN PARA
  ESCRIBIR(factorial)
FIN
```

Ventajas: lenguaje independiente, fácil de entender y depurar.

3.2 Diagramas de flujo

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

3.3 Notaciones tabulares

- Tablas de decisión para visualizar reglas complejas.

3.4 Diagramas Nassi-Shneiderman

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

3.5 Lenguajes de especificación formal

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

3.6 Técnicas modernas

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

4. Metodología de diseño de algoritmos

4.1 Planteamiento del problema

Preguntas clave:

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

4.2 Representación de datos

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

4.3 Descripción de instrucciones

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

4.4 Verificación y optimización

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

5. Técnicas avanzadas de diseño algorítmico

5.1 Programación dinámica

- Guardar resultados intermedios (memoización).

- Ejemplo clásico: Fibonacci con memorización.

5.2 Algoritmos voraces (Greedy)

- Útiles cuando una solución óptima local conduce a una global.

- Ejemplos: problema del cambio de monedas, algoritmo de Dijkstra.

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

5.4 Divide y vencerás
- Pasos: dividir → resolver recursivamente → combinar.
- Ejemplo: Mergesort, Quicksort.

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

6. Eficiencia algorítmica

6.1 Análisis de complejidad

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

7. Aplicaciones prácticas

7.1 Reutilización de algoritmos
- Patrones comunes reutilizables: búsqueda binaria, ordenación, etc.

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

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

8. Conclusiones

8.1 Comparativa de técnicas

Técnica	Complejidad típica	Uso ideal

Fuerza bruta	O(n!)	Problemas pequeños y simples
Divide y vencerás	O(n log n)	Ordenación, búsqueda
Programación dinámica	O(n²)	Subestructuras óptimas, mochila

8.2 Tendencias futuras

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

Tema 24: Lenguajes de programación: Tipos y características

1. Introducción

- **Lenguaje de programación**: Conjunto de reglas sintácticas y semánticas que permiten escribir instrucciones comprensibles para una máquina.
- **Evolución histórica**: Desde el lenguaje ensamblador hasta lenguajes de alto nivel, pasando por paradigmas imperativos, orientados a objetos y funcionales.
- **Tendencias actuales**:
    - Cloud computing y desarrollo distribuido.
    - Inteligencia artificial y aprendizaje automático.
    - Plataformas low-code y no-code.
    - Programación cuántica emergente.

2. Conceptos básicos

2.1 Elementos de los lenguajes de programación

- **Sintaxis y semántica**: Forma (estructura) y significado de las instrucciones.
  Estructuras de control:
    - Secuencial
    - Condicional (`if`, `switch`)
    - Iterativa (`while`, `for`)
- **Tipos de datos**:
    - Primitivos: enteros, booleanos, flotantes
    - Estructurados: arrays, registros
    - Abstractos: listas, pilas, colas, árboles
- **Paradigmas de programación**

3. Características de un lenguaje de programación

- **Expresividad**: Capacidad para representar algoritmos de forma clara y concisa.
- **Eficiencia**: Aprovechamiento de recursos (CPU, memoria).
- **Portabilidad**: Posibilidad de ejecución en distintas plataformas.
- **Modularidad**: Organización en módulos, funciones o clases reutilizables.
- **Seguridad**: Gestión de errores, control de tipos, aislamiento de memoria.

4. Clasificación de los lenguajes de programación

4.1 Según la cercanía a la máquina

- **Lenguajes de bajo nivel**: ensamblador, C.
- **Lenguajes de alto nivel**: Python, Java.
- **Lenguajes intermedios**: C++, Rust (control de bajo nivel con seguridad).

4.2 Según la evolución tecnológica

- **Clásicos**: C, Fortran.
- **Modernos**: Go, Rust (eficientes y seguros).
- **Cuánticos**: Qiskit (Python), Cirq (Google).

Ejemplo básico en Qiskit:

```
from qiskit import QuantumCircuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()
print(qc)
```

- `h(0)`: crea superposición en el qubit 0.
- `cx(0, 1)`: entrelaza qubit 0 y 1.
- Se mide el sistema completo.

4.3 Según la forma de ejecución

- **Compilados**: C, Rust (transformados a binario).
- **Interpretados**: Python, JavaScript (ejecución línea a línea).
- **Híbridos (bytecode)**: Java, C# (intermedio optimizado).
- **Transpilados**: Traducción entre lenguajes (ej. TypeScript a JavaScript).

4.4 Según el paradigma de programación

- **Imperativo**: Describe cómo se realiza la tarea (C, Java, Python).
- **Declarativo**: Describe qué se desea obtener, sin detallar el cómo.

Ejemplo en SQL:

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

- **Funcional**: Usa funciones puras, evita efectos secundarios (Haskell, Lisp).

Ejemplo en Haskell:

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

- **Lógico**: Programación basada en hechos y reglas (Prolog).

Ejemplo en Prolog:

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

- **Orientado a objetos**: Organiza el código en clases y objetos (Java, C#, Python).
- **Reactivo / Eventos**: Para aplicaciones asíncronas y en tiempo real (Node.js, RxJS).
- **Tiempo real**: Sistemas críticos o embebidos (Ada, C, Rust).

Ejemplo en C (tiempo real):

```
while(1) {
  if (sensor_temperatura() > 50) {
    activar_ventilador();
  }
}
```

5. Procesadores de lengujes

5.1 Traductores

- **Compilador**: Traduce todo el programa antes de ejecutarlo (GCC, javac).
- **Intérprete**: Ejecuta línea a línea (Python, Bash).

5.2 Ensambladores

- Traducen lenguaje ensamblador a código máquina (NASM, MASM).

5.3 Entornos de desarrollo

- **IDEs completos**: Visual Studio, Eclipse, JetBrains.
  Editores ligeros: VS Code, Sublime Text.
- **IDE en la nube**: GitHub Codespaces, Replit.

6. Lenguajes de programación actuales

6.1 C y C++

- Sistemas embebidos, videojuegos, compiladores, sistemas operativos.

6.2 Java

- Aplicaciones empresariales, móviles (Android), backend multiplataforma.

6.3 .NET (C#, Visual Basic)

- Entorno Windows, desarrollo de escritorio y aplicaciones web.

6.4 Python

- Ciencia de datos, inteligencia artificial, automatización, web.

6.5 Desarrollo web

- **Frontend**: JavaScript, TypeScript, HTML, CSS.
- **Backend**: Node.js, PHP, Ruby.
- **Full-stack**: Frameworks como React, Angular, Django.

6.6 Plataformas low-code / no-code

- **Low-code**: Requieren poco código (Mendix, OutSystems).
- **No-code**: Totalmente visuales (Bubble, Power Automate).
- **Ventajas**: Accesibilidad, rapidez.
- **Limitaciones**: Escalabilidad, personalización.

7. Conclusión

- Los lenguajes de programación evolucionan para adaptarse a nuevas necesidades tecnológicas: inteligencia artificial, computación cuántica, desarrollo en la nube.

- Paradigmas clásicos (imperativo, OO, funcional) conviven con nuevas propuestas (low-code, reactivo, cuántico).

- La elección del lenguaje depende del contexto: eficiencia, facilidad de mantenimiento, escalabilidad y comunidad.

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

1. Introducción a la Programación Estructurada

- **1.1. Definición y Principios Básicos**
  - Paradigma de la programación estructurada: organización del flujo de control mediante estructuras bien definidas.
  - Principales componentes: secuencia, selección e iteración.
  - Relación con otros paradigmas: funcional, lógico, orientado a objetos.
- 1.2. Historia y Evolución
  - Origen en la década de 1970 con Edgar Dijkstra y su crítica al spaghetti code.
  - Aportaciones clave: modularización, simplificación, y enfoque en el control de flujo.
- 1.3. Objetivos del Uso de la Programación Estructurada
  - Mejora de la legibilidad y mantenibilidad.
  - Reducción de errores mediante control claro de flujo.
  - Facilita la colaboración en proyectos grandes.
- 1.4. Ventajas
  - Facilitación de la depuración.
  - Reducción de complejidad y mayor facilidad de pruebas unitarias.
  - Reutilización de código mediante funciones y procedimientos.
  - Modularización y mantenimiento a largo plazo.

2. Estructuras Básicas

- **2.1. Secuencial**
  - Ejecución del código línea por línea.
  - Ejemplo práctico: secuencia de asignaciones y operaciones.
  - Importancia de las sentencias de inicialización.
- **2.2. Selección (Alternativas)**
  - If-Else: Comparación, condiciones de igualdad y desigualdad.
  - Switch-Case: Alternativas múltiples y su eficiencia en comparación con if-else anidados.
  - Operador Ternario: Optimización en expresiones condicionales simples.
  - Ejemplos de aplicaciones: validación de entradas, decisiones de flujo.
- **2.3. Iteración (Bucles)**
  - For: Iteración definida, iteraciones en secuencias conocidas.
  - While: Uso en condiciones que no se conocen de antemano.
  - Do-While: Garantizar al menos una ejecución.
  - Break y Continue: Control de flujo en bucles y optimización.
  - Ejemplos de uso: navegación en estructuras de datos, procesamiento de archivos.

3. Funciones y Procedimientos

- **3.1. Diferencias Esenciales**
  - Funciones: Valor de retorno, propiedades de inmutabilidad.
  - Procedimientos: Acciones o efectos secundarios sin valor de retorno.
  - Comparativa: Modularidad en funciones y procedimientos en sistemas grandes.
- **3.2. Parámetros y su Manejo**
  - Por valor: Cópia de la información.
  - Por referencia: Paso directo del valor, modificación de la variable original.
  - Parámetros por defecto: Evitar sobrecarga de funciones.
- **3.3. Ámbito de las Variables**
  - Variables locales y globales: prácticas recomendadas y desventajas de usar variables globales.
  - Ámbito estático: Variables que persisten durante la ejecución del programa.
  - Accesibilidad y riesgos asociados con la manipulación incorrecta del ámbito.
- **3.4. Valores de Retorno**
  - Tipos básicos y complejos.
  - Validación de valores de retorno y su uso para manejar flujos de control.
- **3.5. Recursividad Avanzada**
  - Fundamentos de la recursividad: caso base y caso recursivo.
  - Optimización de recursividad: Tail recursion, optimización en lenguajes modernos.
  - Desventajas: Peligro de desbordamientos de pila, ineficiencia en algunos casos.
- **3.6. Ejemplos de Recursividad Completa**
  - Ejemplo práctico de Fibonacci, recorrido en árboles binarios.
  - Análisis de complejidad recursiva.

4. Control de Flujo Avanzado

- **4.1. Manejo de Excepciones**
  - Introducción a la gestión de errores: diferencia entre errores y excepciones.
  - Try-Catch: Manejo eficiente de excepciones.
  - Finally: Liberación de recursos y gestión de errores críticos.
  - Ejemplo de manejo de excepciones en operaciones de E/S.
- **4.2. Sentencias de Control Adicionales**
  - Goto: Casos de uso controvertidos y mejores alternativas.
  - Assert: Evaluación de condiciones en tiempo de ejecución para asegurar el comportamiento esperado.
- **4.3. Excepciones Personalizadas**
  - Creación y manejo de excepciones propias en lenguajes orientados a objetos.

5. Optimización y Eficiencia en Programación Estructurada

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

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

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

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

7. Tendencias Modernas en Programación Estructurada

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

8. Casos Prácticos y Aplicaciones

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

9. Conclusiones

- **9.1. Relevancia en el Mundo Moderno**
    - La programación estructurada sigue siendo fundamental en la formación de programadores y es clave para proyectos de pequeña y mediana escala.
- **9.2. Adaptabilidad y Flexibilidad**
    - Aunque la programación orientada a objetos ha ganado popularidad, el enfoque estructurado sigue siendo esencial para ciertos tipos de sistemas y es un pilar fundamental para aprender otros paradigmas.

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

1. Introducción

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

2. Fundamentos de la Programación Modular

2.1. ¿Qué es un Módulo?

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

2.2. Cohesión y Acoplamiento

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

2.3. Circulación de Datos entre Módulo

- Los módulos se comunican mediante parámetros de entrada/salida o intercambio de mensajes.

    - Parámetros: Los datos se pasan directamente entre funciones (ej: funciones con parámetros de entrada y salida).

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

3. Diseño de Funciones

3.1. Principios Básicos

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

3.2. Uso de Variables Globales

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

3.3. Comunicación entre Componentes y Módulos

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

4. Recursividad

4.1. Estructura Básica de la Recursividad

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

4.2. Optimización de la Recursividad

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

4.3. Stack Overflow

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

5. Librerías y Gestión de Componentes

5.1. Tipos de Librerías

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

5.2. Buenas Prácticas

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

6. Modularidad en Sistemas Complejos

6.1. Modularidad en Arquitecturas de Componentes

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

6.2. Comunicación entre Módulos

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

6.3. Modularidad en el Frontend

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

7. Conclusión

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

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

1. Introducción

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

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

2. DDL – Lenguaje de definición de datos

Tipos de datos modernos

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

Tablas avanzadas

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

Vistas avanzadas

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

Restricciones modernas

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

3. DML – Lenguaje de manipulación de datos

Manipulación avanzada

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

Optimización de consultas

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

- Caching y precomputación: Aceleran consultas frecuentes

Integración con análisis de datos

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

4. DCL – Lenguaje de control de datos

Seguridad granular

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

5. SQL EMBEBIDO – Aplicaciones complejas

Integración distribuida

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

Aplicaciones modernas

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

6. Alternativas a SQL

NoSQL y NewSQL

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

GraphQL

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

Lenguajes específicos de dominio (DSL)

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

7. Optimización y transpilación

Optimización automática

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

Transpilación entre modelos

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

8. Aplicaciones actuales

Business Intelligence

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

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

Learning Analytics

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

9. Perspectivas futuras

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

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

1. Introducción

Concepto de riesgo, impacto y vulnerabilidad.

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

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

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

2. Servicios de Seguridad Aplicados a los Datos

2.1 Confidencialidad

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

2.2 Integridad

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

2.3 Disponibilidad

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

2.4 Autenticidad y No Repudio

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

3. Técnicas de Protección de Datos

3.1 Cifrado Avanzado

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

3.2 Prevención de Pérdida de Datos (DLP)
- Soluciones DLP: Microsoft Purview, Forcepoint.
- Bloqueo de canales de fuga: USB, correo, SaaS.

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

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

4. Sistemas de Protección de Datos

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

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

4.3 Integridad de Archivos y Sistemas
- Hashes automatizados y firmas para verificación de integridad.
- Control de cambios y monitoreo de archivos críticos.

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

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

- Assertions: Definición de reglas de negocio complejas a nivel de esquema.
- **Niveles de aislamiento** (según ANSI SQL):
  - `READ UNCOMMITTED`: Máxima concurrencia, mínima seguridad.
  - `READ COMMITTED`: Previene lecturas sucias (por defecto en muchos SGBD).
  - `REPEATABLE READ`: Evita lecturas no repetibles.
  - `SERIALIZABLE`: Mayor aislamiento, evita todas las anomalías.
- **Control de concurrencia**:
  - **Bloqueos (Locks)**: `SELECT ... FOR UPDATE`, bloqueos a nivel de fila o tabla.
  - **MVCC (Multiversion Concurrency Control)**: Lecturas consistentes sin bloqueos (PostgreSQL, Oracle).
  - **Deadlocks**: Prevención mediante acceso ordenado y detección automática.
- **Gestión de roles y privilegios personalizados**: Acceso granular por funciones.

- **Cifrado**:
  - **TDE (Transparent Data Encryption)**.
  - **Always Encrypted** para columnas sensibles.
- **Validación de entradas y protección frente a inyecciones SQL**: Uso de ORM y consultas parametrizadas.
- **Restricciones de integridad**:
  - `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `NOT NULL`, `CHECK`.
  - **Assertions**: Validaciones complejas sobre múltiples tablas.
- **Triggers (disparadores)**:
  - Auditoría automática de operaciones.
  - Verificación de reglas de negocio en tiempo real.
- **Alta disponibilidad**:
  - SQL Server AlwaysOn, PostgreSQL + Patroni, MySQL Galera Cluster.
- Protección frente a inyecciones SQL: ORMs, validación de inputs, consultas parametrizadas.

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

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

4.6 Clústeres y Replicación
- Alta disponibilidad activa/pasiva: Failover automático.

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

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

5. Estándares y Legislación

5.1 Estándares Internacionales

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

5.2 Legislación

RGPD y LOPDGDD: Base legal, consentimiento, derechos del usuario.
ENS: Marco normativo en la Administración Pública Española.

5.3 Evaluación de Riesgos y Cumplimiento

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

6. Amenazas a la Seguridad de los Datos

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

7. Seguridad de los Datos en la Nube

7.1 Riesgos Específicos en Cloud

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

7.2 Controles de Seguridad en Cloud

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

8. Formación, Buenas Prácticas y Concienciación

8.1 Políticas Internas

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

8.2 Concienciación y Capacitación

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

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

1. Introducción: Interfaces más allá de las pantallas.

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

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

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

3. Principios y heurísticas de diseño

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

4. Diseño visual moderno

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

5. Tipo de interfaces gráficas de usuario (GUI)

5.1 Web

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

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

5.2 Móvil

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

5.3 Escritorio

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

5.4 Emergentes

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

6. Proceso de diseño UX/UI

6.1 Metodologías

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

6.2 Técnicas

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

7. Desarrollo de interfaces modernas

Web

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

Aplicaciones Móviles

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

Aplicaciones de Escritorio

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

Interfaces para Dispositivos Emergentes

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

8. Tendencias futuras

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

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

1. Introducción

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

2. Servicios de Seguridad

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

2.1 Autenticación y Autorización

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

2.2 Control de Acceso

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

2.3 Cifrado y No Repudio

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

2.4 Auditoría y Registro

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

3. Técnicas de Protección

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

3.1 Segmentación

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

3.2 Bastionado (Hardening)

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

3.3 Prevención de Amenazas

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

- Bloqueo por listas negras y reputación IP.

4. Sistemas de Protección

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

4.1 Cortafuegos (Firewalls)

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

4.2 IDS / IPS

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

4.3 Sistemas de Backup y Recuperación

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

5. Estándares, Normativas y Legislación

5.1 Estándares Internacionales

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

5.2 Legislación y Regulaciones

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

5.3 Evaluación de Riesgos y Cumplimiento

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

6. Amenazas Comunes y Emergentes

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

7. Formación y Concienciación

7.1 Actividades Educativas

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

7.2 Iniciativas de INCIBE (España)

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