

Introducción a la Programación 2

Bienvenidos a este curso avanzado donde exploraremos conceptos fundamentales de la programación que van más allá de lo básico. A lo largo de esta presentación, analizaremos estructuras de datos, algoritmos, paradigmas de programación y otros temas esenciales para cualquier programador. Cada sección incluye actividades prácticas para reforzar el aprendizaje.

Estructuras de Datos: Listas, Pilas, Colas, Matrices y Listas Enlazadas

Lista

Acceso por índices

Ejemplo: Lista de nombres

Pila (LIFO)

Último en entrar, primero en salir

Ejemplo: Deshacer acciones

Cola (FIFO)

Primero en entrar, primero en salir

Ejemplo: Impresión en cola

Matriz

Acceso por fila y columna

Ejemplo: Tablas, imágenes

Lista Enlazada

Acceso nodo → nodo

Ejemplo: Navegación de páginas

Actividades Prácticas

1. Simula una pila con las páginas visitadas en un navegador.
2. Representa una cola de espera en un hospital.

Estas estructuras de datos son fundamentales en la programación y se utilizan para resolver diferentes tipos de problemas. La elección de la estructura adecuada depende del tipo de acceso que necesitemos y de las operaciones que vayamos a realizar con mayor frecuencia. Por ejemplo, si necesitamos acceder rápidamente a elementos por su posición, una lista es ideal; si necesitamos mantener un orden de llegada, una cola es la mejor opción.

Árboles y Grafos

Árboles

- Estructuras jerárquicas: padre, hijos
- Tipos: binarios, AVL, B, binarios de búsqueda
- Aplicaciones: sistemas de archivos, bases de datos

Los árboles son estructuras no lineales que representan jerarquías. Cada elemento (nodo) puede tener varios hijos pero un solo padre, excepto el nodo raíz que no tiene padre.

Grafos

- Compuestos por nodos y conexiones (aristas)
- Tipos: dirigidos, no dirigidos, ponderados
- Aplicaciones: mapas, redes sociales, sistemas de transporte

Los grafos son estructuras más generales que los árboles, donde cada nodo puede conectarse a cualquier otro nodo sin restricciones jerárquicas.

Actividad

Actividades Prácticas

1. Representa un árbol genealógico utilizando la estructura de árbol.
2. Crea un grafo que modele una red de amigos, donde cada persona es un nodo y las amistades son aristas.

Tanto los árboles como los grafos son estructuras de datos avanzadas que permiten modelar relaciones complejas entre elementos. Su comprensión es fundamental para resolver problemas de optimización, búsqueda y representación de datos interconectados.

Notaciones Algorítmicas: O, Ω, Θ



Tabla de complejidades

N otació n	Tipo de eficiencia	Ejemplo
1) $O(n)$	Constante	Acceso directo a un array
n)	Lineal	Búsqueda simple
$O(\log n)$	Logarítmica	Búsqueda binaria
$O(n \log n)$	Log-lineal	QuickSort, MergeSort
$O(n^2)$	Cuadrática	Burbuja, selección

Notación O (Big-O)

Representa el límite superior o peor caso del tiempo de ejecución de un algoritmo. Es la más utilizada para analizar la eficiencia.

Ejemplo: $O(n^2)$ para el algoritmo de ordenamiento burbuja.

Notación Ω (Omega)

Representa el límite inferior o mejor caso del tiempo de ejecución de un algoritmo.

Ejemplo: $\Omega(n)$ para el algoritmo de ordenamiento burbuja (cuando la lista ya está ordenada).

Notación Θ (Theta)

Representa tanto el límite superior como inferior cuando ambos son iguales. Es decir, el caso promedio.

Ejemplo: $\Theta(n \log n)$ para el algoritmo de ordenamiento merge sort.

Actividades Prácticas

- Clasifica los siguientes algoritmos según su notación Big-O: búsqueda binaria, ordenamiento burbuja, acceso a arreglo.
- Explica cuándo conviene usar cada uno de estos algoritmos según su eficiencia.

Las notaciones algorítmicas nos permiten comparar la eficiencia de diferentes algoritmos sin depender de factores específicos como el hardware o el lenguaje de programación. Son herramientas esenciales para el análisis y diseño de algoritmos eficientes.

Gráficas y Tablas de Rendimiento Algorítmico

Curvas de Crecimiento (Representación Mental)



Constante $O(1)$

Plano horizontal. El tiempo de ejecución no depende del tamaño de entrada.

Ejemplo: Acceso a un elemento de un array por índice.



Lineal $O(n)$

Recta ascendente. El tiempo crece proporcionalmente al tamaño de entrada.

Ejemplo: Búsqueda secuencial en un array.



Logarítmica $O(\log n)$

Curva que se aplana. Crece lentamente a medida que aumenta la entrada.

Ejemplo: Búsqueda binaria en un array ordenado.



Cuadrática $O(n^2)$

Curva creciente rápida. El tiempo crece exponencialmente con la entrada.

Ejemplo: Ordenamiento burbuja, inserción.

Actividad

Actividades Prácticas

1. Dibuja estas curvas y ordénalas de más a menos eficiente según su comportamiento con grandes volúmenes de datos.
2. Relaciona algoritmos comunes con su curva de crecimiento correspondiente.

Entender visualmente cómo crecen los tiempos de ejecución de los algoritmos nos ayuda a seleccionar el más adecuado para cada situación. Las gráficas de rendimiento son herramientas fundamentales para comparar algoritmos y predecir su comportamiento con diferentes tamaños de entrada.

Paradigmas de Lenguajes de Programación

Paradigma Imperativo

Basado en secuencias de instrucciones que modifican el estado del programa.

Lenguajes comunes: C, Python

Características: Uso de variables, asignaciones, estructuras de control.

Paradigma Funcional

Basado en funciones puras sin efectos secundarios y sin estado.

Lenguajes comunes: Haskell, F#

Características: Inmutabilidad, funciones de orden superior, evaluación perezosa.

Paradigma Lógico

Basado en reglas y hechos para resolver problemas mediante inferencia lógica.

Lenguajes comunes: Prolog

Características: Declarativo, basado en relaciones, unificación de patrones.

Paradigma Orientado a Objetos

Basado en objetos que encapsulan datos y comportamiento.

Lenguajes comunes: Java, C++

Características: Encapsulación, herencia, polimorfismo, clases y objetos.

17. Clasificación de lenguajes por otros criterios

Actividades Prácticas

1. Clasifica varios lenguajes de programación según su paradigma predominante.
2. Escribe una misma operación (por ejemplo, calcular el factorial) en estilo imperativo y funcional.

Los paradigmas de programación representan diferentes enfoques y filosofías para resolver problemas mediante código. Cada paradigma tiene sus fortalezas y debilidades, y muchos lenguajes modernos son multiparadigma, permitiendo combinar diferentes estilos según las necesidades.

Criterios de Clasificación de Lenguajes

Tipado	Compilación	Propósito
<p>Estático: Los tipos se verifican en tiempo de compilación.</p> <p>Ejemplos: Java, C++, TypeScript</p> <p>Dinámico: Los tipos se verifican en tiempo de ejecución.</p> <p>Ejemplos: Python, JavaScript, Ruby</p>	<p>Compilado: Se traduce completamente antes de ejecutarse.</p> <p>Ejemplos: C, C++, Rust</p> <p>Interpretado: Se traduce y ejecuta línea por línea.</p> <p>Ejemplos: Python, JavaScript, PHP</p>	<p>General: Diseñado para múltiples dominios.</p> <p>Ejemplos: Python, Java, C#</p> <p>Específico: Optimizado para un dominio particular.</p> <p>Ejemplos: SQL, R, MATLAB</p>



Preguntas para Reflexionar

1. ¿Es Python interpretado o compilado? (Respuesta: Técnicamente es interpretado, pero primero se compila a bytecode)
2. ¿Qué lenguajes están diseñados principalmente para scripts? (Ejemplos: Python, JavaScript, Bash)
3. ¿Y para sistemas? (Ejemplos: C, C++, Rust)

Estos criterios de clasificación nos ayudan a entender las características fundamentales de los lenguajes de programación y a seleccionar el más adecuado para cada tipo de proyecto. Muchos lenguajes modernos combinan características de diferentes categorías para ofrecer mayor flexibilidad.

Introducción a Punteros y Lenguaje Máquina

Conceptos Fundamentales



Puntero

Variable que almacena la dirección de memoria de otra variable. Permite acceso directo a la memoria y manipulación eficiente de estructuras de datos.

Ejemplo en C: `int *p = &variable;`



Stack (Pila)

Región de memoria que almacena variables locales y llamadas a funciones. Gestión automática con estructura LIFO.

Uso: Variables locales, parámetros de funciones, dirección de retorno.



Heap (Montículo)

Región de memoria para asignación dinámica. Requiere gestión manual en lenguajes como C.

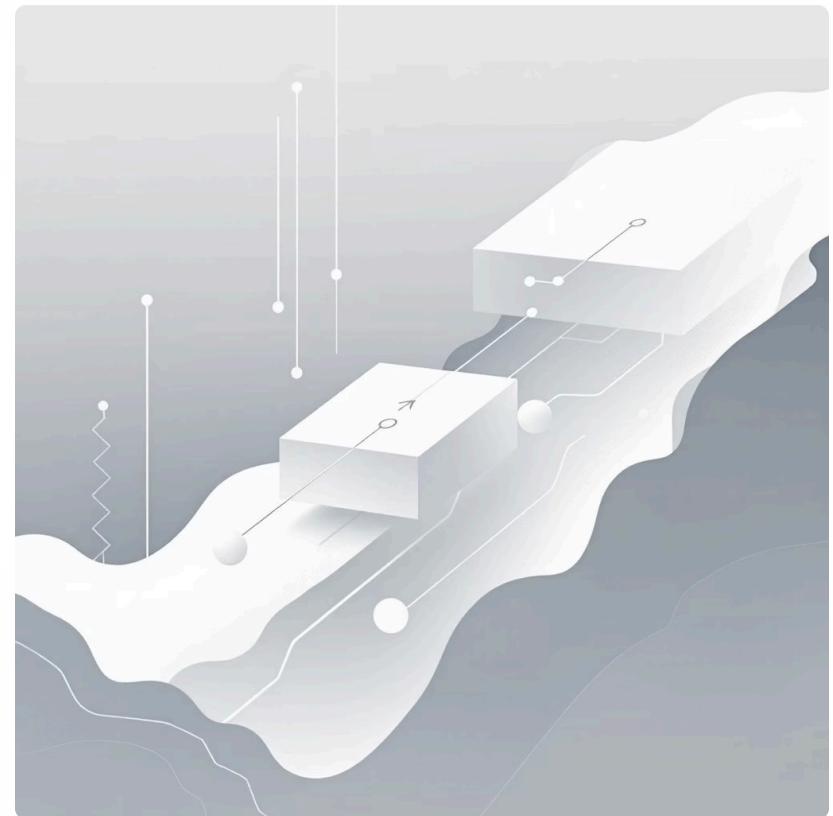
Uso: Objetos, arrays dinámicos, estructuras de datos complejas.



Registros

Variables internas del procesador que almacenan datos temporales durante la ejecución.

Ejemplos: AX, BX, CX, DX en arquitectura x86.



Actividades Prácticas

1. Simula cómo una variable ocupa memoria y cómo un puntero referencia esa dirección.
2. Dibuja el stack y heap durante la ejecución de un programa simple con funciones anidadas.

Entender los punteros y la gestión de memoria es fundamental para programar en lenguajes de bajo nivel como C y C++. Aunque lenguajes de más alto nivel ocultan estos detalles, conocerlos ayuda a comprender el funcionamiento interno de los programas y a optimizar su rendimiento.

Compilación, Linkado y Librerías

Compilación

Proceso de traducir código fuente a código objeto. El compilador analiza la sintaxis, optimiza y genera código máquina.

Resultado: Archivos objeto (.o, .obj)

Linkado

Proceso de unir múltiples archivos objeto y librerías en un ejecutable. Resuelve referencias entre módulos.

Resultado: Archivo ejecutable (.exe, .out)

Librerías

Estáticas (.lib, .a): Se incluyen directamente en el ejecutable durante el linkado.

Dinámicas (.dll, .so): Se cargan en tiempo de ejecución, permitiendo compartir código entre aplicaciones.

Ventajas y Desventajas

Librerías Estáticas

- Ventajas: Ejecutable autónomo, rendimiento optimizado
- Desventajas: Mayor tamaño, actualizaciones requieren recompilación

Librerías Dinámicas

- Ventajas: Menor tamaño, actualizaciones independientes
- Desventajas: Dependencias externas, posibles conflictos de versiones

Actividades Prácticas

1. Compila un programa simple con una librería externa y observa el proceso.
2. Investiga si tu sistema operativo utiliza DLL (Windows) o SO (Linux/macOS).

El proceso de compilación y linkado es fundamental para entender cómo el código fuente se convierte en programas ejecutables. Conocer estos procesos ayuda a resolver problemas de dependencias y a optimizar el rendimiento de las aplicaciones.

Transpilación, Uglify, Minimize y Ofuscación

1

Transpilación

Proceso de convertir código entre lenguajes de programación similares o versiones diferentes del mismo lenguaje.

Ejemplos: TypeScript → JavaScript, ES6+ → ES5, JSX → JavaScript

Herramientas: Babel, TypeScript Compiler

2

Minimización

Proceso de reducir el tamaño de los archivos eliminando espacios, saltos de línea y comentarios sin afectar la funcionalidad.

Beneficios: Menor tiempo de carga, menor ancho de banda

Herramientas: UglifyJS, Terser

3

Uglify

Proceso que hace el código ilegible para humanos, renombrando variables y funciones con nombres cortos y eliminando formato.

Beneficios: Código más compacto, cierta protección contra inspección

Herramientas: UglifyJS, Terser

4

Ofuscación

Técnica para hacer el código deliberadamente difícil de entender, manteniendo su funcionalidad pero ocultando su lógica.

Beneficios: Protección de propiedad intelectual, dificulta ingeniería inversa

Herramientas: JavaScript Obfuscator, ProGuard (Java)

Recursos Recomendados

- Minificador JavaScript: <https://javascript-minifier.com>
- Khan Academy - [Algoritmos](#)
- Visualizador de algoritmos: [Visualgo](#)
- Referencia de algoritmos: [GeeksForGeeks](#)
- Tabla de complejidades: Big-O [Cheat Sheet](#)

Actividades Prácticas

1. Usa <https://javascript-minifier.com> para minificar un fragmento de código JavaScript.
2. Prueba una herramienta de ofuscación y compara el código original con el resultado.

Estas técnicas son fundamentales en el desarrollo web moderno para optimizar el rendimiento y proteger la propiedad intelectual. Entender cómo funcionan te ayudará a implementar mejores prácticas en tus proyectos de desarrollo.