Módulo 1 – Fundamentos de estructuras de datos y análisis de algoritmos

Curso de Introducción a Estructuras de Datos y Algoritmos

21 de agosto de 2025

Contents

1 Módulo 1 – Fundamentos de estructuras de datos y análisis de algoritmos

---

1.1 Objetivos del módulo

Al finalizar este módulo el estudiante será capaz de:

- Definir qué es un algoritmo y describir sus propiedades esenciales.
- Analizar la eficiencia de un algoritmo en tiempo y memoria utilizando notación Big O.
- Reconocer diferentes paradigmas algorítmicos (divide y vencerás, voraces, programación dinámica).
- Explicar qué es un tipo de dato abstracto (TDA) y por qué es independiente de su implementación.
- Comprender el funcionamiento de vectores y matrices como estructuras de datos estáticas.

1.2 Introducción

Cuando hablamos de **algoritmos**, nos referimos a algo tan cotidiano como universal. Un algoritmo no es exclusivo de la informática: es cualquier secuencia **finita y ordenada** de pasos que, si se ejecutan correctamente, conducen a la solución de un problema.

- La receta de una tortilla de patatas es un algoritmo: lista de ingredientes (entrada), pasos de preparación (proceso) y el plato final (salida).
- Seguir las instrucciones para montar un mueble de IKEA es otro algoritmo: si no cumples un paso o lo interpretas mal, el resultado no será el esperado.
- Incluso actividades básicas como calcular mentalmente el cambio al pagar en efectivo son algoritmos en acción.

Lo que distingue a los algoritmos en **ciencias de la computación** es que deben estar formulados de forma tan precisa y sin ambigüedades que **una máquina pueda ejecutarlos**. El ordenador no tiene intuición ni interpreta el contexto como lo haría un humano: si las instrucciones son vagas o incompletas, simplemente no sabrá qué hacer.

1.2.1 La importancia de la eficiencia

Resolver un problema es el primer paso, pero resolverlo de manera **eficiente** es lo que hace que un algoritmo sea útil en la práctica.

Un ejemplo cotidiano:

- Imagina que tienes un diccionario con 100.000 palabras y buscas una en concreto. Puedes leer palabra por palabra desde el inicio (**algoritmo lineal, O(n)**), o usar el índice alfabético para descartar mitades sucesivas (**búsqueda binaria, O(log n)**).

Ambas opciones encuentran la respuesta, pero la segunda lo hace en milésimas de segundo, incluso para volúmenes de datos enormes.

En informática moderna, esta diferencia puede ser la frontera entre lo **viable** y lo **imposible**:

- Google no podría responder a miles de millones de búsquedas diarias sin algoritmos de búsqueda altamente optimizados.
- La secuenciación del genoma humano no habría sido factible en plazos razonables si no se hubieran desarrollado algoritmos y estructuras de datos especializados para manejar grandes volúmenes de información.
- Aplicaciones como Netflix o Spotify, que recomiendan contenidos en tiempo real, dependen de algoritmos capaces de procesar datos en fracciones de segundo.

1.2.2 Estructuras de datos: el soporte de los algoritmos

Los algoritmos no trabajan en el vacío: necesitan **datos** sobre los que operar. Aquí entran en juego las **estructuras de datos**, que son formas organizadas de almacenar y gestionar la información.

Un mismo conjunto de datos puede volverse fácil o difícil de manejar según cómo se organice:

- Si guardas los contactos de tu móvil en una simple lista desordenada, encontrar a alguien concreto requerirá recorrer todos los nombres uno por uno.
- En cambio, si los organizas en un árbol balanceado o en una tabla hash, la búsqueda se convierte en una operación casi instantánea.

Dicho de otro modo: **los algoritmos son los procesos, y las estructuras de datos son el soporte donde esos procesos se ejecutan de manera eficiente**.

1.2.3 Conceptos clave de este módulo

En este módulo introduciremos las **piedras angulares** sobre las que se construye todo el estudio de algoritmos y estructuras de datos:

- **Tipos de Dato Abstracto (TDA)**: nos permiten definir qué operaciones podemos hacer con una estructura (ej. apilar, desapilar en una pila) sin preocuparnos aún de cómo se implementan. Esto favorece la **abstracción** y el diseño modular.
- **Notación Big O**: la herramienta matemática que usamos para expresar la eficiencia de los algoritmos, especialmente cuando el tamaño de los datos crece. No nos

interesa solo saber "cuánto tarda" un programa en un caso concreto, sino prever cómo escalará con miles o millones de datos.
- **Paradigmas algorítmicos**: las "familias" de estrategias que permiten resolver problemas de forma general (divide y vencerás, algoritmos voraces, programación dinámica). Cada paradigma tiene fortalezas y limitaciones que conviene conocer para elegir el enfoque adecuado.
- **Vectores (arrays)**: la primera estructura de datos que estudiaremos. Aunque sencilla, es fundamental para entender la relación entre organización de la memoria, acceso rápido a elementos y coste de operaciones más complejas como inserciones o borrados.

1.2.4 Mirando hacia adelante

Dominar estos fundamentos no es un fin en sí mismo, sino un **punto de partida**. Al finalizar el módulo, el estudiante entenderá que:

- La informática no consiste solo en programar, sino en **diseñar soluciones eficientes**.
- Un problema mal resuelto puede ser inofensivo en pequeña escala, pero absolutamente desastroso cuando los datos se multiplican por millones.
- Los algoritmos y estructuras de datos son el puente entre la teoría matemática y las aplicaciones prácticas: desde buscadores web hasta análisis genómicos, desde videojuegos hasta inteligencia artificial.

Idea clave: Comprender algoritmos y estructuras de datos es aprender a pensar cómo resolver problemas de manera sistemática y óptima. Es adquirir una caja de herramientas que podrás aplicar en cualquier disciplina científica, tecnológica o incluso en la vida cotidiana.

1.3 Concepto y representación de algoritmos

Un **algoritmo** es mucho más que un conjunto de instrucciones: es la **esencia del pensamiento computacional**. Para que un procedimiento pueda considerarse algoritmo, debe cumplir ciertas propiedades fundamentales:

- **Finitud**: el algoritmo debe terminar siempre después de un número finito de pasos. Si no hay garantía de terminación, no estamos ante un algoritmo, sino ante un proceso indefinido. *Ejemplo*: una receta de cocina tiene un final (el plato preparado). En cambio, "remueve indefinidamente la sopa" no cumple este requisito.

- **Definición precisa**: cada paso debe estar claramente especificado y no admitir ambigüedad. Los ordenadores no saben interpretar "un poco de sal" o "mezclar

hasta que esté bien". Necesitan instrucciones exactas.

- **Entrada y salida**: todo algoritmo parte de unos datos de entrada y produce resultados de salida. Sin entrada, el algoritmo carece de sentido; sin salida, no resuelve nada.

Estas propiedades permiten diferenciar los algoritmos de otros procedimientos informales, como consejos, hábitos o descripciones vagas.

1.3.1 Representación de algoritmos

Un algoritmo puede representarse de varias formas, según el contexto y la audiencia:

- **Lenguaje natural**: útil para explicar a principiantes, pero propenso a ambigüedades.
- **Diagramas de flujo**: muy usados en las primeras etapas de diseño; representan gráficamente decisiones, procesos y entradas/salidas.
- **Pseudocódigo**: forma intermedia entre el lenguaje natural y un lenguaje de programación real. Permite expresar ideas con claridad sin preocuparse de la sintaxis exacta.
- **Lenguajes de programación**: cuando el objetivo es la ejecución por ordenador.

```
Algoritmo Maximo(lista[1..n]):
  max ← lista[1]
  para i desde 2 hasta n hacer
    si lista[i] > max entonces
      max ← lista[i]
  devolver max
```

1.3.1.1 Ejemplo: máximo de n números

Este algoritmo recorre toda la lista una sola vez.

- Número de operaciones: proporcional a n.
- Complejidad temporal: $\mathbf{O(n)}$.
- Complejidad espacial: $\mathbf{O(1)}$ (solo una variable adicional `max`).

Este ejemplo sencillo ilustra cómo analizar un algoritmo no solo por su corrección, sino también por su eficiencia.

1.4 Análisis de eficiencia y notación Big O

No todos los algoritmos que resuelven un mismo problema lo hacen con la misma eficiencia. Por eso surge la necesidad de medir y comparar su rendimiento.

La eficiencia se estudia en dos dimensiones principales:

- **Tiempo de ejecución**: cantidad de operaciones realizadas en función del tamaño de la entrada (n).
- **Uso de memoria**: espacio adicional necesario para ejecutar el algoritmo.

Ejemplo motivador:

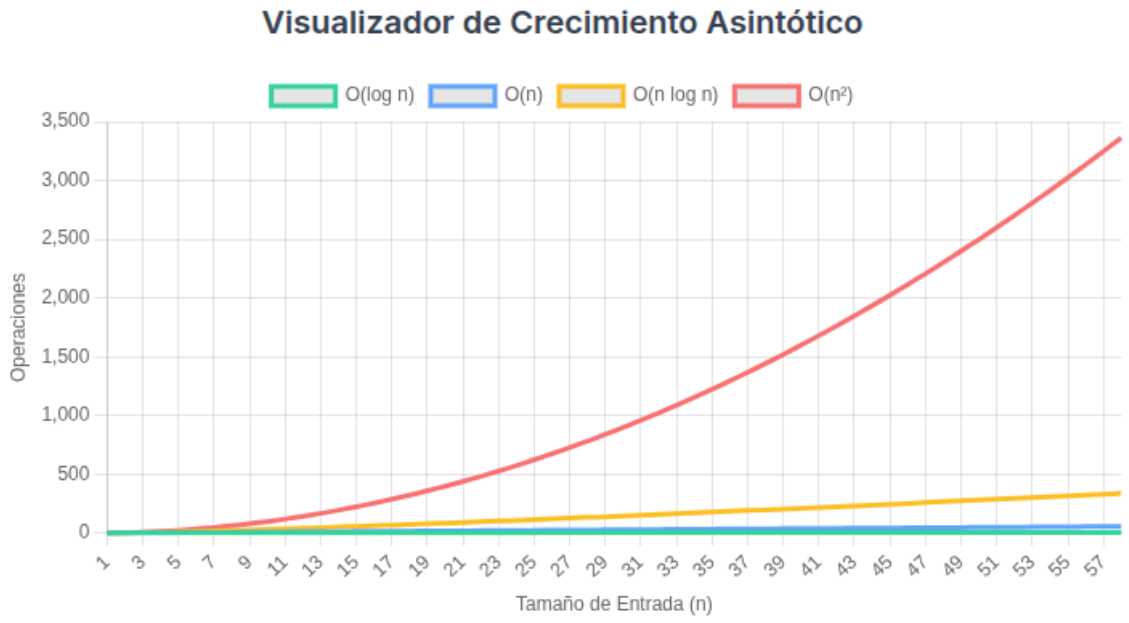
- Ordenar una lista de un millón de elementos con **Bubble Sort ($O(n^2)$)** es impracticable: tardaría horas.
- Con **Merge Sort ($O(n \log n)$)** la misma tarea se resuelve en segundos.

1.4.1 Crecimiento con el tamaño de la entrada

La notación Big O describe el comportamiento asintótico de un algoritmo: cómo crece su tiempo o memoria requerida cuando n se hace grande.

- **$O(n)$**: tiempo lineal. Escala bien con datos grandes.
- **$O(n^2)$**: tiempo cuadrático. Factible solo con volúmenes pequeños.
- **$O(\log n)$**: extremadamente eficiente; un millón de elementos se resuelve en unas 20 operaciones.

Visualización del crecimiento de complejidades comunes:



1.4.2 Peor caso, mejor caso y caso promedio

Analizar un algoritmo no es tan simple como medir un único tiempo:

- **Peor caso**: garantiza que el algoritmo nunca será peor que ese límite. Fundamental en contextos críticos (ej. seguridad informática).
- **Mejor caso**: refleja la situación más favorable (a menudo poco representativa).
- **Caso promedio**: se calcula considerando la distribución estadística de las entradas. Muy útil en práctica real.

Ejemplo:

- **QuickSort** tiene un mejor caso y promedio de **O(n log n)**, pero en el peor caso degenera a **O(n²)**.
- Por eso se aplican estrategias como elegir pivotes aleatorios para evitar los peores escenarios.

1.4.3 Paradigmas algorítmicos

Los algoritmos no se diseñan de manera aislada: suelen seguir **paradigmas de resolución** que sirven como guías generales.

- **Divide y vencerás**: divide un problema en subproblemas más pequeños, resuélvelos y combina resultados. Ejemplo: *Merge Sort*, búsqueda binaria.
- **Voraces (greedy)**: toman decisiones locales óptimas esperando llegar a la solución global. Ejemplo: algoritmo de Kruskal para árboles de expansión mínima.
- **Programación dinámica**: almacena resultados intermedios para no recalcular. Ejemplo: algoritmo de Needleman–Wunsch para alineamiento global de secuencias.

Estos paradigmas son el esqueleto de muchas soluciones modernas en bioinformática, optimización y teoría de grafos.

1.4.4 Tabla de complejidad habitual

Orden	Descripción	Ejemplo
O(1)	Tiempo constante	Acceso a un elemento de un array
O(log n)	Logarítmico	Búsqueda binaria
O(n)	Lineal	Recorrer un vector
O(n log n)	Cuasilineal	*Merge Sort*, *Quicksort*
O(n²)	Cuadrático	*Bubble Sort*
O(2)	Exponencial	Problema de la mochila (fuerza bruta)

Orden	Descripción	Ejemplo
O(n!)	Factorial	Generación de todas las permutaciones

Además de Big O, existen notaciones complementarias:

- **Ω**: cota inferior.
- **Θ**: cota ajustada (cuando superior e inferior coinciden).

1.5 Ejemplos aplicados en bioinformática

La bioinformática es un campo donde la eficiencia algorítmica es crítica:

- **Cálculo del contenido GC** en una secuencia de ADN. Algoritmo lineal **O(n)**.

```
seq = "ATGCGCTAAGC"
gc = sum(1 for base in seq if base in "GC") / len(seq)
print(f"GC%: {gc:.2%}")
```

- **Comparación de todas las parejas de secuencias** en un genoma: **O(n²)**. Impracticable para genomas completos.
- **Alineamiento global**: programación dinámica → **O(n · m)** para dos secuencias de longitudes n y m. Aunque costoso, es mucho más eficiente que la comparación exhaustiva.
- **BLAST**: ejemplo de heurística que sacrifica exactitud a cambio de tiempos cercanos a **O(n)**, lo que lo hizo revolucionario en los años 90.

1.6 Tipos de estructuras de datos

Las estructuras de datos definen cómo se organizan y almacenan los datos en memoria.

1.6.1 Clasificación básica

Tipo	Descripción	Ejemplo
Lineales	Elementos en secuencia	vectores, listas, pilas, colas
Jerárquicas	Relaciones en niveles	árboles, heaps
Grafos	Relaciones generales entre nodos	redes sociales, mapas
Estáticas	Tamaño fijo	arrays
Dinámicas	Tamaño variable	listas enlazadas, árboles, grafos

1.6.2 Tipos de Dato Abstracto (TDA)

Un **TDA** define el comportamiento esperado de una estructura de datos sin entrar en cómo se implementa.

Ejemplo: la **pila** (stack).

- Operaciones: `apilar`, `desapilar`, `consultar`.
- Implementaciones posibles: vector (eficiente en acceso), lista enlazada (eficiente en inserción/eliminación).

Separar concepto de implementación permite diseñar programas más robustos y modulares.

1.7 Vectores y matrices

Los **vectores (arrays)** son la estructura más básica y, a la vez, una de las más poderosas:

- **Acceso aleatorio**: O(1).
- **Recorrido completo**: O(n).
- **Inserciones o eliminaciones en medio**: O(n), por el coste de mover elementos.

Las **matrices** (arrays bidimensionales) permiten representar información tabular o espacial:

- Imagen digital: cada píxel es una celda de la matriz.
- Tabla de expresión genética: filas como genes, columnas como condiciones experimentales.

Ejemplo práctico: recorrer una matriz por filas suele ser más rápido que por columnas porque aprovecha la **localidad de caché** de la CPU. Esto ilustra cómo los detalles de hardware influyen en el rendimiento.

1.8 Conclusiones

En esta parte hemos:

- Definido los requisitos de un algoritmo.
- Visto cómo representarlos en pseudocódigo y diagramas de flujo.
- Aprendido a medir su eficiencia con Big O y otras notaciones.
- Exploramos paradigmas algorítmicos y su relevancia en problemas reales.
- Introducimos la clasificación de estructuras de datos y el concepto de TDA.

- Introducimos vectores y matrices como estructuras estáticas fundamentales.

Entender estos conceptos es adquirir el lenguaje básico con el que se construyen todos los sistemas informáticos modernos.

1.9 Ejercicios de autoevaluación

1. Explica con tus palabras por qué un algoritmo debe ser finito. Da un ejemplo de procedimiento no finito.
2. Representa en pseudocódigo un algoritmo que busque el mínimo de una lista. ¿Cuál es su complejidad temporal?
3. ¿Qué diferencia práctica hay entre un algoritmo O(n) y otro O(n log n) para $n = 1.000.000$?
4. Explica por qué QuickSort se considera eficiente a pesar de tener un peor caso $O(n^2)$.
5. ¿Cómo implementarías una pila con un vector? ¿Y con una lista enlazada? ¿Qué ventajas e inconvenientes observas?
6. Diseña un ejemplo donde recorrer una matriz por filas sea más eficiente que por columnas y explica por qué.

1.10 Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms*. MIT Press.
- Sedgewick, R., & Wayne, K. *Algorithms*. Addison-Wesley.
- Goodrich, M. T., & Tamassia, R. *Data Structures and Algorithms in Java*. Wiley.
- Kleinberg, J., & Tardos, É. *Algorithm Design*. Pearson.
- Gusfield, D. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press (bioinformática).