Módulo 7 – Algoritmos de ordenación

Curso de Introducción a Estructuras de Datos y Algoritmos

21 de agosto de 2025

Contents

1 Módulo 7 – Algoritmos de Ordenación: Imponiendo el Orden en el Caos

1.1 Motivación: El Orden como Prerrequisito de la Eficiencia

En un mundo saturado de datos, la información en su estado crudo es a menudo caótica e inmanejable. La **ordenación** es el proceso fundamental mediante el cual transformamos este caos en una estructura inteligible. Es, quizás, la tarea más realizada en computación, no como un fin en sí misma, sino como un paso crucial que habilita operaciones más complejas y eficientes.

La filosofía detrás de la ordenación es simple: **invertir trabajo ahora para ahorrar mucho más trabajo después**.

- **Búsqueda eficiente**: Como ya vimos, sin orden, encontrar un elemento requiere una búsqueda lineal ($O(n)$). Con orden, la búsqueda binaria lo encuentra en tiempo logarítmico ($O(\log n)$). Esta es la diferencia entre encontrar un libro en una biblioteca

desorganizada frente a una catalogada por el sistema decimal Dewey.

- **Análisis de datos**: ¿Cuál es el valor mediano de un conjunto? ¿Cuáles son los elementos duplicados? ¿Qué valores están en el percentil 99? Estas preguntas son triviales de responder sobre datos ordenados, pero computacionalmente costosas sobre datos desordenados.
- **Fundamento algorítmico**: Muchos algoritmos avanzados presuponen que los datos de entrada están ordenados. Desde encontrar los dos puntos más cercanos en un plano hasta la compresión de datos, la ordenación es el primer paso indispensable.

El estudio de los algoritmos de ordenación es un viaje por la historia de la algoritmia. Nos enseña sobre diferentes enfoques para resolver un mismo problema (fuerza bruta, divide y vencerás) y nos obliga a pensar en los *trade-offs* fundamentales: tiempo vs. memoria, simplicidad vs. eficiencia, y el comportamiento en el mejor, peor y caso promedio.

1.2 Algoritmos Simples (Complejidad Cuadrática)

Estos algoritmos son conceptualmente sencillos y fáciles de implementar, pero su rendimiento de $O(n^2)$ los hace inviables para conjuntos de datos que no sean pequeños. Son, sin embargo, excelentes herramientas pedagógicas para entender los fundamentos de la ordenación.

1.2.1 Bubble Sort (Ordenación por Burbuja)

- **Filosofía**: La ”fuerza bruta” paciente. Compara repetidamente pares de elementos adyacentes y los intercambia si están en el orden incorrecto. En cada pasada completa, el siguiente elemento más grande ”burbujea” hasta su posición final.
- **Analogía**: Imagina una fila de personas de diferentes alturas. En cada paso, miras a dos personas contiguas y, si la de la izquierda es más alta que la de la derecha, las intercambias. Si repites este proceso a lo largo de toda la fila suficientes veces, la gente terminará ordenada por altura.

```
procedimiento bubbleSort(lista):
    n ← longitud(lista)
    hacer
        intercambiado ← falso
        para i desde 0 hasta n-2:
            si lista[i] > lista[i+1]:
                intercambiar(lista[i], lista[i+1])
```

```
                intercambiado ← verdadero
        n ← n - 1 // Optimización: el último elemento ya está en su sitio
    mientras intercambiado
```

1.2.1.1 Pseudocódigo

- **Análisis**: Su rendimiento es pobre ($O(n^2)$) porque solo mueve los elementos de uno en uno. Su única ventaja real es su capacidad de detectar si la lista ya está ordenada (terminando en una sola pasada, $O(n)$).

1.2.2 Insertion Sort (Ordenación por Inserción)

- **Filosofía**: Construir el orden de forma incremental. Recorre la lista, tomando cada elemento y "deslizándolo" hacia la izquierda en la parte ya ordenada de la lista hasta encontrar su lugar correcto.
- **Analogía**: Es exactamente como la mayoría de la gente ordena una mano de cartas. Tomas una carta a la vez y la insertas en la posición correcta entre las cartas que ya tienes ordenadas en la otra mano.

```
procedimiento insertionSort(lista):
    para i desde 1 hasta longitud(lista)-1:
        clave ← lista[i]
        j ← i - 1
        // Desplazar elementos mayores que la clave hacia la derecha
        mientras j ≥ 0 y lista[j] > clave:
            lista[j+1] ← lista[j]
            j ← j - 1
        lista[j+1] ← clave
```

1.2.2.1 Pseudocódigo

- **Análisis**: Aunque su peor caso sigue siendo $O(n^2)$, es significativamente más eficiente en la práctica que Bubble Sort. Su gran ventaja es su rendimiento **adaptativo**: para listas que están **casi ordenadas**, su complejidad se acerca a $O(n)$, lo que lo hace muy útil en ciertos escenarios.

1.2.3 Selection Sort (Ordenación por Selección)

- **Filosofía**: El método metódico. Divide la lista en dos partes: una ordenada (al principio) y una desordenada (el resto). En cada paso, encuentra el elemento más pequeño de la parte desordenada y lo intercambia con el primer elemento de esa parte, expandiendo así la sección ordenada.

- **Analogía**: Es como un director de casting que tiene que poner en fila a un grupo de actores por altura. Primero, busca al actor más bajo de todo el grupo y lo pone al principio. Luego, ignora a esa persona y busca al más bajo del resto, colocándolo en la segunda posición. Repite hasta que todos están en fila.

```
procedimiento selectionSort(lista):
    n ← longitud(lista)
    para i desde 0 hasta n-2:
        índice_mínimo ← i
        para j desde i+1 hasta n-1:
            si lista[j] < lista[índice_mínimo]:
                índice_mínimo ← j
        intercambiar(lista[i], lista[índice_mínimo])
```

1.2.3.1 Pseudocódigo

- **Análisis**: Su complejidad es **siempre** $O(n^2)$, sin importar el estado inicial de la lista. Su principal característica es que minimiza el número de intercambios, lo que podría ser útil si la operación de intercambio es muy costosa.

1.3 Algoritmos Eficientes

Complejidad $O(n \log n)$

Estos algoritmos utilizan estrategias más sofisticadas, típicamente basadas en el paradigma **"Divide y Vencerás"**, para lograr una eficiencia muy superior. Son el estándar de oro para la ordenación de propósito general.

1.3.1 Merge Sort (Ordenación por Mezcla)

- **Filosofía**: La organización recursiva. La idea es que es trivial ordenar una lista de un solo elemento. Merge Sort divide recursivamente la lista a la mitad hasta que solo quedan sublistas de un elemento. Luego, combina (fusiona o "merge") esas sublistas de manera ordenada hasta reconstruir la lista completa.
- **El paso clave**: La función `fusionar(izquierda, derecha)` es el corazón del algoritmo. Toma dos sublistas ya ordenadas y las combina en una nueva lista ordenada en tiempo lineal $O(n)$.

```
función mergeSort(lista):
    si longitud(lista) ≤ 1:
```

```
        devolver lista

    medio ← longitud(lista) / 2
    izquierda ← sublista(lista, 0, medio-1)
    derecha ← sublista(lista, medio, fin)

    izquierda_ordenada ← mergeSort(izquierda)
    derecha_ordenada ← mergeSort(derecha)

    devolver fusionar(izquierda_ordenada, derecha_ordenada)

procedimiento fusionar(izquierda, derecha):
    // Código para combinar dos listas ordenadas en una nueva
    ...
```

1.3.1.1 Pseudocódigo

- **Análisis**: Su complejidad es **siempre** $O(n \log n)$, lo que lo hace muy predecible y fiable. Su principal desventaja es que requiere espacio adicional ($O(n)$) para almacenar las sublistas, lo que puede ser un problema con memoria limitada. Es un algoritmo **estable**, lo que significa que mantiene el orden relativo de los elementos con claves iguales.

1.3.2 Quicksort (Ordenación Rápida)

- **Filosofía**: La partición inteligente. Es otro algoritmo de "Divide y Vencerás", pero funciona de manera diferente.
  1. **Elegir un pivote**: Se selecciona un elemento de la lista (el pivote).
  2. **Particionar**: Se reorganiza la lista de modo que todos los elementos menores que el pivote queden a su izquierda, y todos los mayores queden a su derecha.
  3. **Recursión**: Se aplica Quicksort recursivamente a las dos sublistas (la de los menores y la de los mayores).
- **El paso clave**: La eficiencia de Quicksort depende críticamente de la elección del pivote. Un buen pivote divide la lista en dos mitades de tamaño similar. Un mal pivote (el menor o mayor elemento) puede degradar el rendimiento a $O(n^2)$.

```
función quickSort(lista):
    si longitud(lista) ≤ 1:
        devolver lista

    pivote ← elegir_pivote(lista)
```

```
    menores, iguales, mayores ← particionar(lista, pivote)

    resultado ← concatenar(quickSort(menores), iguales, quickSort(mayores
    devolver resultado
```

1.3.2.1 Pseudocódigo (Conceptual)

- **Análisis**: A pesar de su peor caso de $O(n^2)$, su rendimiento **promedio** es $O(n \log n)$ con constantes muy bajas, lo que lo hace extremadamente rápido en la práctica. Funciona *in-place*, requiriendo solo $O(\log n)$ de espacio en la pila de recursión. Es el algoritmo de ordenación de propósito general más utilizado en las bibliotecas estándar de muchos lenguajes.

1.4 Comparación y Criterios de Elección

Algoritmo	Complejidad Promedio	Peor Caso	Espacio Adicional	Estable	Comentarios
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(1)$		Solo para fines e
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$		Eficiente en lista
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$		Predecible pero
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$		Muy fiable. Idea
Quicksort	$O(n \log n)$	$O(n^2)$	$O(\log n)$		Generalmente el

¿Qué algoritmo elegir?

- Para **listas pequeñas** (ej. < 20 elementos), **Insertion Sort** suele ser el más rápido.
- Para una **garantía de rendimiento** y si la **estabilidad** es importante, **Merge Sort** es la elección segura.
- Para el **máximo rendimiento promedio** en memoria, **Quicksort** es el rey.
- Para **datos masivos** que no caben en memoria, una variante de **Merge Sort** (ordenación externa) es la única opción.

¿Para qué se usan en computación y bioinformática?

- **Insertion sort**: ordenar pequeñas listas (ej. registros temporales).
- **Merge sort**: ordenación externa, como ordenar archivos genómicos de varios gigabytes en disco.
- **Quicksort**: ordenación interna en memoria, ampliamente usado en librerías estándar, siendo el algoritmo por defecto en muchas librerías de C, Java o Python.
- **Ordenación en bioinformática**:
  - Clasificación de secuencias de ADN/proteínas antes de alineamientos masivos.

- Preparación de grandes matrices de expresión génica para análisis estadísticos.
- Preprocesamiento de datos de lecturas en secuenciación masiva.

1.5 Conclusiones

- Los algoritmos **cuadráticos** son simples, útiles solo para listas pequeñas o fines pedagógicos.
- **Merge sort** y **quicksort** ofrecen rendimiento $O(n \log n)$ y son los más usados en la práctica.
- La **estabilidad** y el **uso de memoria** son criterios importantes en la elección.
- La ordenación es clave para optimizar procesos posteriores como búsquedas y análisis de grandes volúmenes de datos.

1.6 Ejercicios de autoevaluación

1. Ordena manualmente la lista `[5,2,9,1,5,6]` con bubble sort, mostrando cada pasada.
2. ¿Por qué insertion sort es más eficiente que bubble sort en la práctica?
3. Explica por qué merge sort es estable y selection sort no lo es.
4. Analiza el peor caso de quicksort. ¿Cómo puede mitigarse?
5. Diseña un algoritmo que combine **insertion sort** y **quicksort** para aprovechar lo mejor de ambos.
6. ¿Qué algoritmo usarías para ordenar un archivo de 50 GB almacenado en disco? Justifica tu respuesta.

1.7 Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms.* MIT Press.
- Weiss, M. A. *Data Structures and Algorithm Analysis.* Pearson.
- Sedgewick, R., & Wayne, K. *Algorithms.* Addison-Wesley.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. *Data Structures and Algorithms in Java.* Wiley.