Introducción a los Algoritmos de Ordenamiento (Sorting)

Dr. Matías Blaña

Departamento de Informática
Universidad Técnica Federico Santa María

Clase simulada – Postulación Académica 2025

Motivación

- Ordenar datos es una de las tareas más frecuentes en programación y análisis de datos.
- Ejemplos:
  - Ordenar registros en una base de datos.
  - Clasificar archivos por tamaño o fecha.
  - Preparar datos antes de aplicar algoritmos de búsqueda o análisis.
  - Clasificar u ordenar partículas, galaxias, o mediciones experimentales por alguna propiedad.
- Objetivo: transformar una lista `[5, 2, 9, 1]` en `[1, 2, 5, 9]`.

Tipos de algoritmos de ordenamiento

- **Simples:** Bubble Sort, Selection Sort, Insertion Sort. (Brute Force).
- **Eficientes:** Merge Sort, Quick Sort, Heap Sort.
- **Especializados:** Counting Sort, Radix Sort, Bucket Sort.
- **Algoritmos avanzados:** Timsort, Bitonic Sort (GPU), IntroSort.

¿Cuántos algoritmos existen?
La literatura computacional describe más de **30 algoritmos de ordenamiento clásicos**, y más de **100 variantes y optimizaciones** (según la clasificación de Wikipedia 2025).

Idea general: ordenar comparando, dividiendo o combinando datos.

Ejemplo: Bubble Sort (simple pero secuencial)

Lista inicial a ordenar:

$$A = [\,\mathbf{5},\ \mathbf{1},\ 4,\ 2,\ 8\,] \Rightarrow [\,1,\ \mathbf{5},\ \mathbf{4},\ 2,\ 8\,] \Rightarrow [\,1,\ 4,\ \mathbf{5},\ \mathbf{2},\ 8\,]$$

$\Rightarrow [\,1,\ 5,\ 4,\ 2,\ 8\,] \Rightarrow [\,1,\ 4,\ \mathbf{5},\ \mathbf{2},\ 8\,] \Rightarrow \ldots$

Idea: recorrer la lista repetidamente, intercambiando pares de elementos adyacentes mal ordenados.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Complejidad: Promedio: $O(n^2)$.

Difícil de paralelizar: cada iteración depende del resultado anterior.

Ejemplo: Insertion Sort (bueno para listas pequeñas)

Idea: insertar cada elemento en su posición correcta dentro de la parte ya ordenada.

$$A = [\,\mathbf{5},\ \mathbf{1^*},\ 4,\ 2,\ 8,\ 2\,] \Rightarrow [\,1,\ \mathbf{5},\ \mathbf{4^*},\ 2,\ 8,\ 2\,] \Rightarrow [\,1,\ 4,\ \mathbf{5},\ \mathbf{2^*},\ 8,\ 2\,]$$

$$\Rightarrow [\,1,\ 4,\ \mathbf{2},\ \mathbf{5},\ 8,\ 2\,] \Rightarrow [\,1,\ \mathbf{2},\ \mathbf{4},\ 5,\ 8,\ 2\,] \Rightarrow [\,1,\ 2,\ 4,\ 5,\ \mathbf{8},\ \mathbf{2}\,]$$

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
```

Complejidad: $O(n^2)$, pero eficiente para listas cortas o casi ordenadas.
Difícil de paralelizar: la inserción depende del orden previo.

Ejemplo: Merge Sort

$$A = [5,\ 1,\ 4,\ 2,\ 8]$$

Idea:

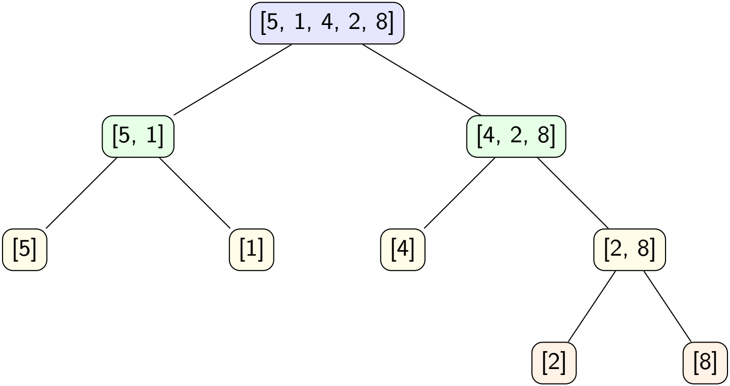
- Dividir la lista en mitades.
- Ordenar cada mitad recursivamente.
- Combinar (merge) las mitades ordenadas.

Complejidad: $O(n \log n)$ (constante).
Paralelización natural: cada mitad puede procesarse en distintos hilos o núcleos.

Ejemplo: Merge Sort

Ejemplo visual:

$$A = [5,\ 1,\ 4,\ 2,\ 8]$$



Fusión final: $[5,1] \to [1,5], \quad [2,8] \to [2,8], \quad [1,5] \cup [4] \cup [2,8] \to [1,2,4,5,8]$

Ejemplo: Timsort, un algoritmo híbrido moderno

Idea principal: Combina las ventajas de **Merge Sort** y **Insertion Sort**. Diseñado por *Tim Peters* (2002) para Python y Java.

Estrategia:

1. Divide el arreglo en segmentos pequeños llamados **runs**.
2. Cada run se ordena con **Insertion Sort**.
3. Luego se combinan los runs con **Merge Sort**.

Ventajas:

- Detecta secuencias ya ordenadas (*natural runs*).
- Muy eficiente para datos parcialmente ordenados.
- Estable y con complejidad garantizada: $O(n \log n)$.

Uso actual: Timsort es el método por defecto en `sorted()` y `list.sort()` en Python.

Quick Sort (divide y conquista)

Ejemplo visual: elección de pivote y partición

$$A = [\,5,\ 1,\ 4,\ 2,\ 8\,]$$

1. Elegimos un pivote: $p = 4$

$$[\,\mathbf{5},\ \mathbf{1},\ \mathbf{4},\ \mathbf{2},\ \mathbf{8}\,] \Rightarrow \begin{cases} \text{Menores: } [\,1,\ 2\,] \\ \text{Iguales: } [\,4\,] \\ \text{Mayores: } [\,5,\ 8\,] \end{cases}$$

2. Aplicar QuickSort recursivamente en cada parte:

$$[\,1,\ 2\,] \Rightarrow [\,1,\ 2\,] \qquad [\,5,\ 8\,] \Rightarrow [\,5,\ 8\,]$$

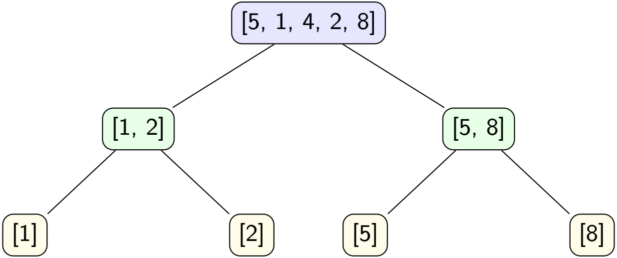
3. Combinar las partes:

$$[\,1,\ 2\,] \cup [\,4\,] \cup [\,5,\ 8\,] \Rightarrow [\,1,\ 2,\ 4,\ 5,\ 8\,]$$

QuickSort: Árbol de recursión

Ejemplo visual con pivote central:

$$A = [\,5,\ 1,\ 4,\ 2,\ 8\,], \qquad p = 4$$



Resultado final:

$$[\,1,\ 2,\ 4,\ 5,\ 8\,]$$

Clave: cada rama es independiente ⇒ **paralelizable**.

Ejemplo: Quick Sort (divide y conquista)

Resumen Idea:

1. Elegir un *pivote*.
2. Dividir el arreglo en menores y mayores al pivote.
3. Ordenar recursivamente cada mitad.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left  = [x for x in arr if x < pivot]
    mid   = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + mid + quick_sort(right)
```

Complejidad promedio: $O(n \log n)$
Altamente paralelizable: cada sublista se procesa en paralelo.

Complejidad, eficiencia y tiempo de cómputo

Algoritmo	**Mejor**	**Promedio**	**Peor**
Bubble / Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$

Complejidad, eficiencia y tiempo de cómputo

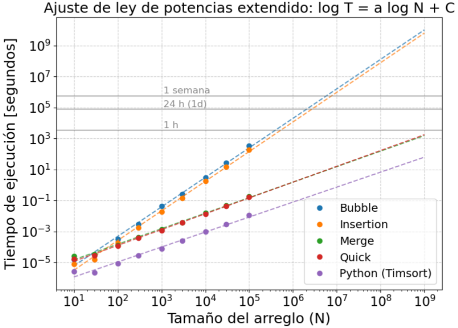
Objetivo: conectar la teoría de complejidad con el tiempo real de ejecución.

Modelo de ajuste:

$$\log T = a \, \log N + C \quad \Rightarrow \quad T \propto N^a$$

Resultados:

- Los algoritmos eficientes (**Merge**, **Quick**, **Timsort**) siguen $a \approx 1.0{-}1.2$.
- La extrapolación muestra el crecimiento del tiempo hasta semanas para $N \sim 10^8$.
- Diferencias pequeñas en a implican grandes variaciones en tiempo real.



Escalamiento temporal y eficiencia de algoritmos de ordenamiento. Test in 1CPU Apple M1 Pro, python Python 3.13.5

Conclusión: La complejidad asintótica ($O(n^2)$, $O(n \log n)$) predice la tendencia, pero el rendimiento real depende de constantes, cache, y paralelización.

¿Cuáles algoritmos se pueden paralelizar?

Algoritmo	Paralelizable	Comentarios
Bubble / Insertion	$\times$	Iterativos, dependencias secuenciales
Merge Sort	✓	Cada mitad es independiente
Quick Sort	✓	Cada sublista se procesa en paralelo
Heap Sort	$\triangle$	Parcialmente (construcción del heap)
Counting / Radix	✓	Ideal para GPU / procesamiento masivo

Conclusión: Los métodos de *divide y conquista* escalan mejor en arquitecturas paralelas.

Comparación de algoritmos

Algoritmo	Promedio	Memoria	Paralelizable
Bubble / Insertion	$O(n^2)$	Baja	$\times$
Merge Sort	$O(n \log n)$	Alta	$\checkmark$
Quick Sort	$O(n \log n)$	Media	$\checkmark$
Heap Sort	$O(n \log n)$	Baja	$\triangle$

En práctica: QuickSort es el estándar en librerías (C, Python, C++) por su eficiencia promedio.

Visualización: paralelización en Merge y Quick Sort

Idea general:

- Los algoritmos *divide y conquista* crean subproblemas independientes.
- Cada sublista puede ser procesada en paralelo (en distintos núcleos o GPUs).
- Ejemplo: **MergeSort paralelo**
  - Paso 1: dividir en dos mitades.
  - Paso 2: ordenar cada mitad en paralelo.
  - Paso 3: combinar resultados.

Herramientas comunes: `OpenMP`, `MPI`, `CUDA`, `multiprocessing (Python)`.

Aplicaciones en ingeniería

- Búsqueda binaria y estructuras de datos ordenadas.
- Ordenamiento previo a operaciones de fusión de bases de datos.
- Optimización en pipelines de Big Data (MapReduce, Spark).
- Procesamiento paralelo en GPU (Radix Sort, Bitonic Sort).

Ejemplo: ordenar 10^6 registros distribuidos antes de combinarlos en clúster.

Bibliografía y recursos recomendados

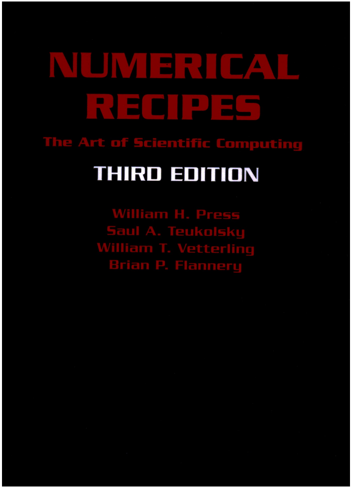
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009).
Introduction to Algorithms (3rd ed.). MIT Press.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007).
Numerical Recipes: The Art of Scientific Computing (3rd ed.).
Cambridge University Press.

Sedgewick, R., & Wayne, K. (2011).
Algorithms (4th ed.). Addison-Wesley.

Wikipedia contributors. (2025).
Sorting algorithm. Recuperado de
`https://en.wikipedia.org/wiki/Sorting_algorithm`

GeeksforGeeks. (2025).
Sorting Algorithms Tutorials. Recuperado de
`https://www.geeksforgeeks.org/sorting-algorithms/`



Portada de *Numerical Recipes* (3rd Ed., 2007)

Conclusión

- Ordenar datos es un paso fundamental en sistemas informáticos.
- Existen distintos algoritmos con costos y estrategias diversas.
- Los enfoques **divide y conquista** (Merge, Quick) son los más eficientes.
- En la era multicore y GPU, la **paralelización** es esencial.

¡Gracias!