Introducción a los Algoritmos de Ordenamiento (Sorting)

Dr. Matías Blaña
Astrofísico, Universidad de La Serena

Clase simulada – Postulación Académica 2025

Departamento de Informática
Universidad Técnica Federico Santa María

¿Qué es sorting, y por qué es importante?

Sorting es el proceso de **reorganizar una lista** para que sus elementos queden en un orden definido (creciente, decreciente, alfabético, etc.).

Ejemplo sencillo:

$$[5,\ 2,\ 9,\ 1] \implies [1,\ 2,\ 5,\ 9]$$

El ordenamiento es clave en la informática moderna.

Presente en casi todo sistema que usamos:

- Bases de datos
- Compresores (e.g. ZIP, LZ77)
- Motores de búsqueda y ranking
- Sistemas distribuidos y cómputo masivo

Mensaje clave: Mucho más que estudiar algoritmos aislados — aprender sorting es entender cómo **razonar sobre eficiencia, escalabilidad y arquitectura**.

El punto no es memorizar algoritmos, sino saber elegir el adecuado según el contexto.

¿Cómo elegir el algoritmo adecuado?

No existe el mejor algoritmo. Depende del contexto:

Escenario	**Algoritmo recomendado**
Datos **casi ordenados**	**Timsort** (e.g. Python). Detecta “runs”.
Poca memoria disponible	**Heap Sort**. No usa memoria extra.
Grandes volúmenes en disco	**Merge Sort**. Clásico en bases de datos.
Tiempo real en ventanas pequeñas	**Insertion Sort**. Eficiente para arreglos chicos.
Promedio rápido en memoria RAM	**Quick Sort**. Excelente localidad de caché.

Idea central: elegir el método según arquitectura, memoria, y distribución de los datos.

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, IntroSort.

¿Cuántos algoritmos existen?
La literatura computacional describe más de **20 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:

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

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

Idea: recorrer la lista repetidamente, intercambiando pares de elementos adyacentes mal ordenados.
Y en pseudo código:

```
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.

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

```
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:

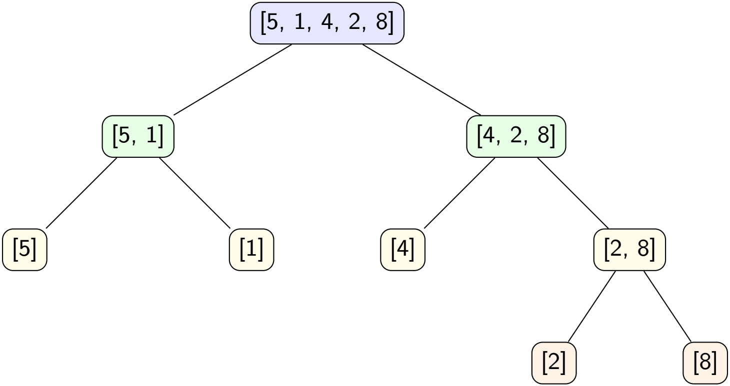
- Dividir la lista en mitades.
- Ordenar cada mitad recursivamente.
- Combinar (merge) las mitades ordenadas.

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.

Ejemplo: QuickSort in-place: idea general

QuickSort in-place reordena el arreglo dentro del mismo arreglo, sin crear listas nuevas.

Pasos principales:

1. Elegir un pivote (aquí usamos el valor **4**).
2. Usar dos punteros:
   - i avanza desde la izquierda buscando valores **mayores** al pivote.
   - j retrocede desde la derecha buscando valores **menores** al pivote.
3. Cuando $i < j$, se realiza un **swap**.
4. Al terminar: el pivote queda en su posición final y el arreglo queda dividido en:

$$\text{menores} \mid \text{pivote} \mid \text{mayores}$$

Ventaja clave: usa menos memoria y es muy rápido en práctica.

QuickSort in-place: partición con *i* y *j*

Arreglo inicial:

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

1. i avanza ($\rightarrow$) hasta encontrar un valor > 4:

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

2. j retrocede ($\leftarrow$) hasta encontrar un valor < 4:

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

3. Cuando $i < j$, se intercambian:

$$A \;\Rightarrow\; [\,2,\, 1,\, 4,\, 5,\, 8\,]$$

4. i avanza nuevamente:

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

   i llega al pivote $\Rightarrow$ STOP.

Resultado de la partición:

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

QuickSort in-place: resultado de la partición

Después de la fase de partición:

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

El pivote (4) está en su posición definitiva.

Ahora QuickSort se aplica recursivamente a cada lado:

$$\text{QuickSort}([2,1]) \Rightarrow [1,2]$$

$$\text{QuickSort}([5,8]) \Rightarrow [5,8]$$

Lista final ordenada:

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

Ventajas:

- In-place: menos uso de memoria.
- Eficiente en promedio: $O(n \log n)$.

Complejidad: ¿Qué significa realmente $O(n)$, $O(n^2)$, $O(n \log n)$?

La notación Big-O describe cómo crece el <u>número de operaciones</u> que realiza un algoritmo cuando el tamaño del arreglo n aumenta.

- $O(n)$: el número de operaciones crece linealmente con n.
- $O(n^2)$: si duplico n $(2n)$, el trabajo se multiplica por **4**.
- $O(n \log n)$: crece más rápido que lineal, pero mucho más lento que cuadrático.

Ejemplo: Bubble Sort, tiene dos ciclos anidados

$$\text{operaciones} \propto n \times n = n^2$$

Por eso decimos que su complejidad es $O(n^2)$.

Importante:

- Big-O describe operaciones abstractas (comparaciones, asignaciones...)
- El tiempo real depende además de CPU, caché, implementación, etc.
- Pero ambos están directamente relacionados: más operaciones $\rightarrow$ más tiempo.

Complejidad empírica: número de operaciones

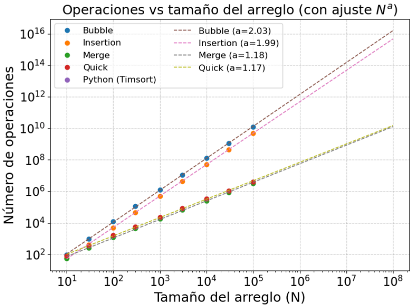
Objetivo: medir cómo crece la cantidad total de operaciones (comparaciones + asignaciones) al aumentar el tamaño del arreglo N.

Modelo ajustado:

$$\log(\text{ops}) = a\log(N) + C \qquad \Rightarrow \qquad \text{ops} \propto N^a$$

Interpretación:

- Bubble/Insertion: exponentes $a \approx 2$ (cuadráticos).
- Merge/Quick: $a \sim 1{-}1.2$, consistente con $O(n\log n)$.
- El ajuste empírico reproduce fielmente la teoría.
- La extrapolación a $N = 10^7 - 10^8$ permite estimar el costo real.



Número de operaciones vs tamaño del arreglo, con ajuste N^a.

Nota: Timsort no se muestra en operaciones porque no es accesible desde Python para instrumentación, pero sí aparece en los gráficos de tiempo.

Complejidad, eficiencia y tiempo de cómputo

La complejidad también dependerá del ordenamiento inicial de los arreglos. Esto significa que se debe seleccionar el algoritmo más optimo o eficiente **dependiendo del problema en particular y de los recursos disponibles**.

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

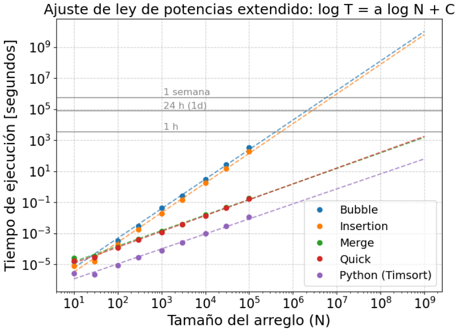
Objetivo: conectar la teoría de complejidad con el tiempo real de ejecución.

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	✓
Quick Sort	$O(n \log n)$	Media	✓
Heap Sort	$O(n \log n)$	Baja	$\triangle$

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

Sorting en aplicaciones reales de alto rendimiento

Aplicaciones: En cómputo científico y análisis de datos masivos:

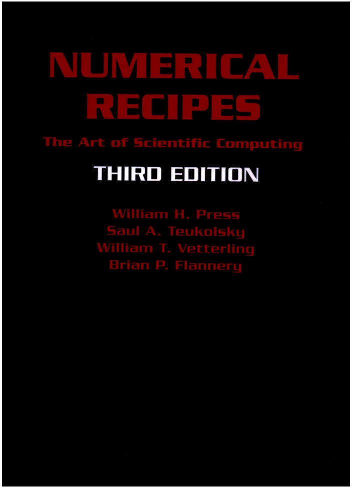
- Ordenar catálogos astronómicos de millones de objetos
- Preparar datos para análisis en paralelo (MPI, GPU)
- Agrupar y clasificar datos multidimensionales
- Pre-procesamiento para algoritmos de clustering y ML

En la práctica: Elegir entre $O(n^2)$ y $O(n \log n)$ no es académico: puede significar **segundos vs horas** o **horas vs días**.

Mi experiencia: en proyectos de simulación y análisis astronómico, optimizar el sorting reduce el tiempo de pipelines completos.

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/`

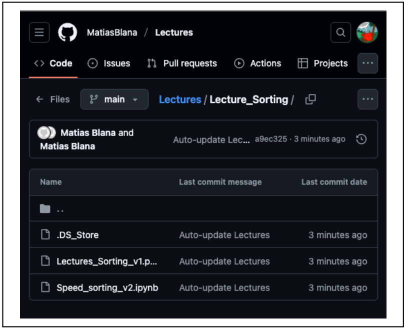


Portada de *Numerical Recipes* (3rd Ed., 2007)

Material de la clase de Sorting

Repositorio del curso:

- Slides en PDF
- Scripts de benchmark (Python)
- Figuras y gráficos
- Ejemplos de algoritmos (Bubble, Merge, Quick, Timsort)



Captura del repositorio en GitHub

Disponible en GitHub:

https://github.com/MatiasBlana/Lectures/tree/main/LectureSorting

Takeaway

El ordenamiento no es un fin, sino una herramienta.

- Los algoritmos de sorting ilustran cómo razonar sobre **eficiencia, escalabilidad y arquitectura**.
- La complejidad teórica ($O(n^2)$, $O(n \log n)$) se refleja directamente en el rendimiento real, como vimos en los benchmarks.
- Cada algoritmo es óptimo en distintos contextos: memoria limitada, datos parcialmente ordenados, GPU, grandes volúmenes en disco, o sistemas distribuidos.
- Timsort es un ejemplo moderno de ingeniería algorítmica: híbrido, estable, adaptativo y usado en varios lenguajes (e.g. Python).
- Entender estas ideas permite construir sistemas más rápidos, eficientes y robustos.

Muchas gracias.

*Material y código disponible en GitHub.

Apéndice: QuickSort in-place: pseudocódigo clásico

QuickSort aplica: partición → recursión → concatenación implícita

```
function quicksort(A, low, high):
    if low < high:
        p = partition(A, low, high)
        quicksort(A, low, p)
        quicksort(A, p+1, high)
```

Partición in-place (Hoare) para el ejemplo con pivote $= 4$:

```
function partition(A, low, high):
    pivot = A[low]      # = 4
    i = low - 1
    j = high + 1

    while true:
        repeat:
            i = i + 1
        until A[i] >= pivot

        repeat:
            j = j - 1
        until A[j] <= pivot

        if i >= j:
            return j

        swap(A[i], A[j])
```

Resultado en este caso:

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