UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Integrador - Programación I

Título del trabajo: “Análisis de Algoritmos de Ordenamiento**”**

Alumnos:

- Ezequiel Alejandro Ventura - equimdq@gmail.com
- Abel Tomás Romero - abeltomasr98@gmail.com

Materia: Programación I

Profesora: Prof. Cinthia Rigoni

Tutor: Prof. Walter Pintos

Fecha de Entrega: 09/06/2025

Índice

1. Introducción

Se desarrolló un programa en Python cuyo objetivo principal es analizar y comparar el rendimiento de distintos algoritmos de ordenamiento. El programa permite al usuario elegir el tamaño y tipo de lista a ordenar (aleatoria, ordenada o inversamente ordenada), así como los algoritmos de ordenamiento a utilizar. Seguidamente, se mide el tiempo de ejecución de cada algoritmo con el fin de evaluar su eficiencia en diferentes contextos. Esta comparación nos facilita a la comprensión del comportamiento de los algoritmos en función del tipo de datos y escalabilidad, dando una base práctica para el estudio de la complejidad algorítmica.

2. Marco Teórico

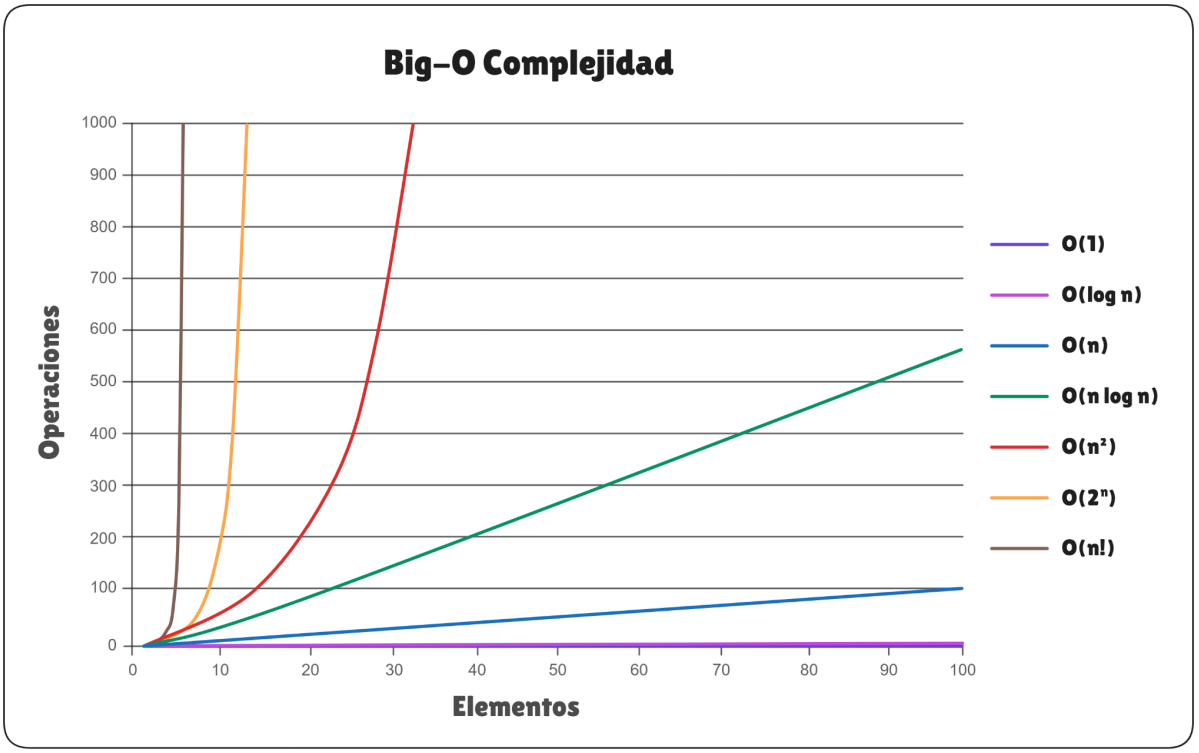
2.1 ¿Qué es un algoritmo de ordenamiento?

Un algoritmo de ordenamiento es un conjunto de instrucciones que permiten organizar los elementos de una lista según un criterio dado. Los algoritmos de ordenamiento son fundamentales en la informática ya que optimizan búsquedas, mejoran el acceso a datos y es muy habitual encontrarlos en sistemas y aplicaciones.

Existen distintos algoritmos con diferentes estrategias, lo que los lleva a tener diferentes niveles de eficiencia y la elección del más adecuado va ligada al contexto y al tipo de dato a procesar.

2.2 Complejidad algorítmica

La complejidad algorítmica permite saber el costo de ejecutar un algoritmo, en función del tamaño de la entrada (n). Se expresa usando notación Big O:



- **O(1):** Constante, el algoritmo tarda lo mismo sin importar la cantidad de datos.
- **O(log n):** Logarítmico, el algoritmo reduce el problema a la mitad en cada paso.
- **O(n):** Lineal, el tiempo de ejecución crece en proporción directa al tamaño de la entrada “n”. Básicamente si se duplican los datos, se duplica también el tiempo.
- **O(n log n):** Log-lineal, es una mezcla de “n” (cada dato se procesa al menos una vez) y “log n” (se aplica división o agrupación eficiente). Más eficiente que “O(n^2)” para grandes volúmenes.
- **O(n^2):** Cuadrática, el tiempo de ejecución crece en proporción al cuadrado del tamaño de entrada. Si se duplican los datos, el tiempo se cuadruplica.
- **O(2^n):** Tiempo exponencial, cada nuevo dato duplica el trabajo
- **O(n!):** Tiempo factorial, el algoritmo evalúa todas las permutaciones posible de datos.

Un algoritmo puede comportarse de forma diferente según el caso:

- **Mejor caso:** situación más favorable.
- **Peor caso:** situación más costosa.
- **Caso promedio:** comportamiento esperado en la mayoría de los casos.

2.3 Algoritmos seleccionados

1. **Selection Sort**
    - Selecciona el elemento más pequeño y lo coloca en su posición correcta, repitiendo este proceso para los elementos restantes.
    - Es fácil de implementar y no requiere memoria adicional. Pero es ineficiente para listas grandes y no es estable.
    - Complejidad: $O(n^2)$ en todos los casos.
2. **Insertion Sort**
    - Construye la lista ordenada insertando cada elemento en su posición correcta dentro de la parte ya ordenada.
    - Es eficiente para listas pequeñas o casi ordenadas, y es estable. La desventaja es que su rendimiento disminuye con listas grandes.
    - Complejidad: Mejor caso → $O(n)$; promedio y peor caso → $O(n^2)$
3. **Bubble Sort**
    - Compara pares adyacentes y los intercambia si están en el orden incorrecto, repitiendo este proceso hasta que la lista esté ordenada.
    - Muy sencillo de entender e implementar, pero es uno de los algoritmos más ineficientes para listas grandes.
    - Complejidad: Mejor caso → $O(n)$;promedio y peor caso → $O(n^2)$
4. **Quick Sort**
    - Utiliza la estrategia de “divide y vencerás”, ya que selecciona un pivote y particiona la lista en sublistas de elementos menores y mayores que el pivote, aplicando recursivamente el mismo proceso.
    - Muy eficiente en la mayoría de los casos y no requiere memoria adicional significativa. Su desventaja es que su rendimiento puede degradarse si el pivote no se elige adecuadamente, y no es estable.
    - Complejidad: Mejor y promedio → $O(n \log n)$; peor caso → $O(n^2)$
5. **Merge Sort**
    - Divide la lista en mitades, ordena cada mitad y luego las combina en una lista ordenada.
    - Consistente y estable, con su rendimiento buen garantizado, pero requiere memoria adicional para las sublistas.
    - Complejidad: Todos los casos → $O(n \log n)$

6. **Tim Sort**
    - Algoritmo híbrido que combina “Insertion Sort” y “Merge Sort”, utilizando por defecto en Python y Java.
    - Es muy eficiente para listas reales que suelen tener patrones parcialmente ordenados y es estable. Aunque su implementación es más compleja.
    - Complejidad: Mejor caso → O(n); promedio y peor caso → O(n log n)

Tabla de complejidad temporal

Algoritmo	Mejor Caso	Caso Medio	Peor Caso
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Tim Sort	O(n)	O(n log n)	O(n log n)

3. Caso Práctico

3.1 Descripción del problema a resolver

Se desarrolló un sistema interactivo en Python que permite comparar el rendimiento de algoritmos de ordenamiento en diferentes escenarios. El problema central que aborda este proyecto es la necesidad de evaluar empíricamente cuál algoritmo de ordenamiento es más eficiente según el tamaño y tipo de datos a procesar. El programa resuelve la problemática mediante:

- Permite al usuario elegir dos algoritmos y observar sus diferencias de rendimiento.
- Evalúa el comportamiento en listas aleatorias, ordenadas e invertidas.
- Prueba con diferentes tamaños de listas (1.000, 10.000 y 30.000 elementos).
- Proporciona tiempos de ejecución exactos y porcentajes de diferencia.

3.2 Arquitectura, organización del código y decisiones de diseño

El proyecto se estructuró de forma modular para facilitar el mantenimiento y la comprensión del mismo:

- `funciones/`	Subdirectorios de funciones del proyecto.
├── `algoritmos.py`	Implementación de los 6 algoritmos.
├── `comparador.py `	Lógica de medición y comparación.
├── `generador_listas.py`	Generación de datos de prueba.
└── `infos.py`	Información y documentación.
- `main.py`	Punto de entrada del programa.

- **Medición de tiempo:** Se utiliza `time.time()` para capturar el tiempo antes y después de la ejecución, dándonos una precisión de microsegundos.
- **Gestión de memoria:** Cada algoritmo opera sobre una copia independiente de la lista original usando `lista.copy()`.
- **Interfaz de usuario:** Se implementó un sistema de menús intuitivo que guía al usuario paso a paso, incluyendo validación de entradas.
- **Modularidad:** La separación de módulos permite reutilizar componentes y facilita futuras extensiones del programa.

3.3 Código fuente comentado

A continuación se presentan algunas de las funciones más importantes del sistema:

Función principal (main.py)

```
def main():
    while True:
        mostrar_menu()
        opcion = input("\nIngrese una opción (1-3): ").strip()

        match opcion:
            case "1":···
            case "2":···
            case "3":···
            case _:···
```

La función `main()` implementa un bucle principal del programa utilizando la estructura `match-case`, coordinando todo el flujo del programa, desde la presentación del menú hasta la gestión de opciones del usuario.

Función de comparación de algoritmos (comparador.py)

```python
def comparar_algoritmos():
    """ Compara dos algoritmos de ordenamiento seleccionados por el usuario. """
    print("\n" + "="*65)
    print("========  COMPARACIÓN DE DOS ALGORITMOS DE ORDENAMIENTO  ========")
    print("" + "="*65)

    while True:
        # Elegir algoritmos
        alg1, alg2 = elegir_algoritmos()

        # Mostrar información de los algoritmos seleccionados
        mostrar_info_algoritmos(alg1, alg2)

        # Generar lista
        lista_original = generar_lista()

        # Crear copias de la lista para cada algoritmo
        copia1 = lista_original.copy()
        copia2 = lista_original.copy()

        print("Ejecutando algoritmos...\n")

        # Algoritmo 1
        inicio1 = time.time()
        nombre1 = aplicar_algoritmo(alg1, copia1)
        fin1 = time.time()
        tiempo1 = fin1 - inicio1
        print(f"• Algoritmo 1 ({nombre1}): Ejecutado en {tiempo1:.4f} segundos")

        # Algoritmo 2
        inicio2 = time.time()
        nombre2 = aplicar_algoritmo(alg2, copia2)
        fin2 = time.time()
        tiempo2 = fin2 - inicio2
        print(f"• Algoritmo 2 ({nombre2}): Ejecutado en {tiempo2:.4f} segundos")

        # Comparación
        if tiempo1 < tiempo2: ···
        elif tiempo2 < tiempo1: ···
        else: ···
```

Esta función es el núcleo del programa, ejecuta todas las funciones que se encargan de:

- Gestionar la selección de algoritmos por parte del usuario.
- Generar las listas de prueba según las especificaciones.
- Crear copias independientes para cada algoritmo.
- Medir los tiempos de ejecución con precisión.
- Calcular y mostrar las diferencias de rendimiento.

Algoritmos de ordenamiento implementados (algoritmos.py)

```python
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        indice_min = i
        for j in range(i+1, n):
            if lista[j] < lista[indice_min]:
                indice_min = j
        lista[i], lista[indice_min] = lista[indice_min], lista[i]
    return lista
```

```python
def insertion_sort(lista):
    for i in range(1, len(lista)):
        temp = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > temp:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = temp
    return lista
```

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

```python
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    pivot = lista[len(lista) // 2]
    izq = [x for x in lista if x < pivot]
    medio = [x for x in lista if x == pivot]
    der = [x for x in lista if x > pivot]
    return quick_sort(izq) + medio + quick_sort(der)
```

```python
def merge_sort(lista):
    if len(lista) <= 1:
        return lista
    mid = len(lista) // 2
    izq = merge_sort(lista[:mid])
    der = merge_sort(lista[mid:])
    return merge(izq, der)

def merge(izq, der):
    resultado = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1
    resultado.extend(izq[i:])
    resultado.extend(der[j:])
    return resultado
```

```python
def tim_sort(lista):
    return sorted(lista)
```

Generador de listas de prueba (generador_listas.py)

```python
def generar_lista():
    """ Genera una lista de prueba según las opciones del usuario """
    print("\n----- Tamaño de la lista -----")
    print("  1. Pequeña (1.000 elementos)")
    print("  2. Mediana (10.000 elementos)")
    print("  3. Grande (30.000 elementos)")

    while True:
        tam_op = input("\nSeleccione el tamaño: ").strip()
        if tam_op == "1":
            n = 1000
            break
        elif tam_op == "2":
            n = 10000
            break
        elif tam_op == "3":
            n = 30000
            break
        else:
            print("Opción no válida. Intente nuevamente.")
```

```python
print("\n----- Tipo de lista -----")
print("  1. Aleatoria")
print("  2. Ordenada")
print("  3. Ordenada invertida")

while True:
    tipo_op = input("\nSeleccione el tipo: ").strip()
    if tipo_op in ["1", "2", "3"]:
        match tipo_op:
            case "1":
                lista = generar_lista_aleatoria(n)
            case "2":
                lista = generar_lista_ordenada(n)
            case "3":
                lista = generar_lista_ordenada_inver(n)

        print(f"\nLista generada con {n} elementos.")
        print(f"Inicio: {lista[:5]} ... {lista[-5:]} Fin.\n")
        return lista
    else:···
```

Esta función permite al usuario seleccionar el tamaño y tipo de lista para las pruebas. Incluye validación de entrada y genera diferentes tipos de listas.

Sistema de información (infos.py)

```python
def mostrar_instrucciones():···


# Información de los algoritmos de ordenamiento
info_algoritmos = {
    1: {
        "nombre": "SelectionSort",
        "descripcion": "Recorre toda la lista buscando el mínimo y colocándolo en su posición correcta.",
        "ventajas": "Simple de implementar, sin estructura auxiliar.",
        "desventajas": "Ineficiente en listas grandes.",
        "complejidad": "O(n²) en todos los casos."
    },
    2: {···
    3: {···
    4: {···
    5: {···
    6: {···
}
```

Contiene la información detallada sobre cada algoritmo, incluyendo descripción, ventajas, desventajas y complejidad temporal, toda esta información se muestra al usuario antes de ejecutar las comparaciones. Y también muestra las instrucciones de uso del programa.

NOTA: Para ver todo el código fuente completo del proyecto, incluyendo todos los módulos y funciones auxiliares, está disponible en el repositorio de GitHub referenciado en la sección de “Anexos” de este documento.

3.4 Validación del funcionamiento

El programa fue validado mediante:

- La verificación del ordenamiento, porque se confirmó que todos los algoritmos producen listas correctamente ordenadas.
- La consistencia en las mediciones, ya que las diferencias de tiempo son coherentes con la complejidad teórica de cada algoritmo aplicado.
- Su robustez, que le permite al programa manejar diferentes tamaños y tipos de listas sin errores.

4. Metodología utilizada

La siguiente metodología nos permitió desarrollar este programa que cumple con los objetivos que nos planteamos, para poder proporcionar una herramienta práctica para el análisis comparativo de algoritmos de ordenamiento.

4.1 Investigación previa

El desarrollo comenzó con una investigación sobre todos los algoritmos de ordenamiento, consultando diferentes fuentes para comprender tanto desde las estructuras de datos nativas, el análisis de complejidad temporal y espacial de cada algoritmo, y sus implementaciones con sus mejores prácticas.

4.2 Etapas de desarrollo

1. **Estructura inicial y arquitectura del proyecto**
   Primero establecimos el repositorio y la configuración básica del proyecto. Luego definimos la estructura y arquitectura modular del programa, definiendo las funciones principales y la organización de archivos que facilitaría el desarrollo colaborativo.

2. **Funcionalidad core y menú principal**
   Ajustamos la estructura y arquitectura inicial basándose en las primeras pruebas, estableciendo el menú general y la lógica para comparar

específicamente dos algoritmos. Luego implementamos gradualmente la interfaz para que el usuario pueda elegir los algoritmos a comparar.

3. **Generación de datos de prueba**
   Luego desarrollamos el sistema de generación de listas con diferentes características (aleatoria, ordenada, invertida) y múltiples tamaños (1.000, 10.000, 30.000 elementos). También agregamos con eso nuevo, más mensajes claros para la comunicación con el usuario, así el programa sea más intuitivo.

4. **Sistema de medición y comparación**
   Se implementó la comparación de tiempos de los algoritmos, para medir el rendimiento de cada uno con las diferentes tipos de listas que tiene disponibles. Y con eso se ajustó un poco más la lógica general del proyecto para evitar errores a la hora de comparar.

5. **Expansión de algoritmos**
   Una vez hecho las anteriores funcionalidades, completamos la implementación de los seis algoritmos de ordenamiento y con eso se agregó el cálculo de porcentajes para mostrar no solo la diferencia en tiempo, sino también el porcentaje de mejora entre algoritmos.

6. **Finalización y documentación**
   Terminando el proyecto, se implementaron mensajes educativos que muestran características, ventajas y desventajas de cada algoritmo antes de la comparación. También se refinó los mensajes del programa y se creó el archivo README con las instrucciones completas de uso y descripción del proyecto.

4.3 Herramientas y recursos utilizados

Durante el desarrollo del trabajo se utilizaron las siguientes herramientas:

- **Visual Studio Code:** Como entorno de desarrollo principal para la escritura y prueba del código.
- **Python 3.10+:** Lenguaje utilizado para implementar y medir el rendimiento de los algoritmos.

- **Módulos estándar:** `time` para la medición del tiempo de ejecución y `random` para la generación de datos aleatorios.
- **Git/GitHub:** Tanto para el seguimiento de cambios y colaboración, y para el repositorio remoto para su documentación.
- **Google Docs y Google Slides:** Para la redacción colaborativa tanto del pdf como de las diapositivas.
- **Discord:** Plataforma de comunicación empleada para coordinar tareas y resolver todas las dudas en equipo.

4.4 Trabajo colaborativo

Nos organizamos de manera conjunta e individual con trabajos pequeños cada uno, para avanzar con el proyecto lo más rápido posible. Juntos trabajamos en el desarrollo del código, la integración de las diferentes funciones y sus pruebas, el refinamiento de la experiencia de usuario, la redacción tanto de documentación del proyecto, de este pdf y las diapositivas para el video.

5. Resultados obtenidos

Los resultados obtenidos a partir de las pruebas muestran diferencias significativas en la eficiencia de los algoritmos de ordenamiento evaluados. En general, **Tim Sort**, que es utilizado por defecto en Python, demostró ser el más rápido en prácticamente todos los casos, gracias a su capacidad para detectar patrones preexistentes y combinar técnicas de Merge Sort e Insertion Sort. Le siguió **Merge Sort**, con un rendimiento constante y predecible, mientras que **Quick Sort** también presentó buenos resultados, especialmente en listas aleatorias, aunque su desempeño puede degradarse si el pivote no se selecciona adecuadamente.
Para visualizar estas diferencias, se presenta a continuación una tabla comparativa que resume los tiempos estimados de ejecución (en segundos) para cada algoritmo al ordenar listas tanto de 1.000 y 30.000 elementos bajo tres escenarios distintos: lista aleatoria, lista ordenada y lista inversamente ordenada. Los valores reflejan el tiempo promedio tras múltiples ejecuciones:

Lista de 1.000 elementos

Algoritmo	Lista Aleatoria (s)	Lista Ordenada (s)	Lista Inversa (s)
Selection Sort	0.0192	0.0173	0.0180
Insertion Sort	0.0200	0.0001	0.0363
Bubble Sort	**0.0380**	**0.0244**	**0.0469**
Quick Sort	0.0012	0.0011	0.0010
Merge Sort	0.0019	0.0013	0.0014
Tim Sort	**0.0001**	**0.0000**	**0.0000**

Lista de 30.000 elementos

Algoritmo	Lista Aleatoria (s)	Lista Ordenada (s)	Lista Inversa (s)
Selection Sort	14.7846	15.0025	16.4747
Insertion Sort	17.4451	0.0037	37.0489
Bubble Sort	**36.9575**	**20.5211**	**48.1056**
Quick Sort	0.0536	0.0376	0.0377
Merge Sort	0.0733	0.0455	0.0476
Tim Sort	**0.0039**	**0.0006**	**0.0005**

Los algoritmos con complejidad cuadrática (**Selection, Insertion y Bubble Sort**) mostraron una fuerte dependencia del orden inicial de los datos y del tamaño de la lista, con escaladas de tiempo significativas, especialmente en listas inversamente ordenadas.

Insertion Sort mostró buen rendimiento en listas ordenadas, acercándose a algoritmos más eficientes en ese caso particular. Por su parte, **Quick Sort** y **Merge Sort** mantuvieron tiempos estables en todos los escenarios, con ventaja para Quick Sort en todos los tipos de listas.

Tim Sort fue el algoritmo más rápido en todos los casos evaluados, demostrando su eficiencia adaptativa y su justificación de por qué es el algoritmo por defecto en Python.

6. Conclusiones

Este trabajo fue una oportunidad para aplicar e integrar de forma práctica muchos de los conceptos vistos en la materia. A través del desarrollo del proyecto, el grupo pudo comprender mejor cómo funcionan internamente los algoritmos de ordenamiento, sus tiempos de ejecución y su comportamiento frente a distintos tipos y tamaños de datos. También reforzamos temas como la complejidad algorítmica, la modularización del código y el uso eficiente de estructuras y funciones en Python.

Uno de nuestros mayores aprendizajes fue cómo trasladar la teoría a un sistema funcional, creando un programa que permite experimentar con distintas variables y visualizar las diferencias de rendimiento entre algoritmos. Esto nos resultó útil no sólo para entender la eficiencia de los algoritmos, sino también para futuros proyectos donde requiera trabajar con grandes volúmenes de datos o tomar decisiones sobre rendimiento.

En cuanto al trabajo en equipo, fue importante dividir tareas, mantener una comunicación constante y resolver problemas en conjunto. Surgieron dificultades como la elección del diseño de un menú flexible, la validación de entradas y básicamente cómo llevar nuestra idea en código, sin que contenga fallos, pero se resolvieron mediante consultas, pruebas y una buena organización del código.

Con posibles mejoras futuras, podría integrarse una interfaz gráfica para facilitar el uso del programa por parte de usuarios no técnicos, agregar más algoritmos para comparar o incluso guardar los resultados obtenidos para futuros análisis. También sería interesante aplicar técnicas de visualización para representar gráficamente el proceso de ordenamiento.

En resumen, este proyecto fortaleció habilidades de programación, análisis y trabajo colaborativo, y nos brindó más experiencia para el desarrollo de futuros sistemas más complejos.

7. Bibliografía

- Panthema. (2013). *The Sound of Sorting – "Audibilization" and Visualization of Sorting Algorithms*. https://panthema.net/2013/sound-of-sorting/
- Nate Gentil. (2025). *¿Cómo piensa tu PC? Algoritmos de ordenación y complejidad temporal (Big O) explicados* [Video]. YouTube. https://www.youtube.com/watch?v=TnTu1QWaWc0&t=1076s
- Wikipedia contributors. (s.f.). *Algoritmo de ordenamiento*. Wikipedia. https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento
- VisuAlgo. (s.f.). *Sorting Algorithms*. VisuAlgo. https://visualgo.net/en/sorting
- Myphz. (2021). *Sort Visualizer.* https://sortvisualizer.com/
- Bigocheatsheet. (s.f.). *Big-O Cheat Sheet*. https://www.bigocheatsheet.com/

8. Anexo

8.1 Evolución histórica de los algoritmos de ordenamiento

Desde los primeros días de la informática, el problema de ordenar datos ha sido clave para optimizar búsquedas, facilitar el análisis y mejorar el rendimiento de los sistemas. Los primeros algoritmos de ordenamiento, como **Bubble Sort**, **Selection Sort**, e **Insertion Sort**, fueron diseñados por su simpleza y facilidad de implementación, ideales para aprender estructuras algorítmicas y lógica de programación.

Sin embargo, su **complejidad cuadrática** los hace hoy en día ineficientes para grandes volúmenes de datos. A medida que crecieron las necesidades de análisis, surgieron algoritmos más eficientes. Fue así que en los años 60 se introdujeron métodos como **Merge Sort** (creado por ***John von Neumann*** en 1945) y **Quick Sort** (inventado por ***Tony Hoare*** en 1959), que mejoraron drásticamente el tiempo de

ejecución en la gran mayoría de los casos ya que introdujeron enfoques más avanzados como *“divide y vencerás”*, reduciendo la complejidad a O(n log n).
Con el tiempo, y en la medida que los lenguajes de programación y los procesadores evolucionaron, surgieron algoritmos híbridos como **Tim Sort** (creado por ***Tim Peters*** en 2002), que combina Merge Sort e Insertion Sort, y es el algoritmo de ordenamiento por defecto en lenguajes como **Python** y **Java**. Este tipo de algoritmo moderno está optimizado para datos reales, aprovechando patrones que ya existían al recorrer listas.

El estudio de la evolución de estos algoritmos, muestra una clara tendencia: buscar un equilibrio entre **simplicidad**, **velocidad** y **adaptabilidad**. Comprender esta evolución ayuda no sólo a elegir el algoritmo adecuado, sino también a apreciar el ingenio detrás del diseño algorítmico y su impacto en la informática moderna.

8.2 Vídeo explicativo en YouTube

https://www.youtube.com/watch?v=xY-uw5Y_U7w

8.3 Repositorio en GitHub

https://github.com/Tomu98/TP-Integrador