Trabajo Práctico Integrador: Algoritmos de Búsqueda y Ordenamiento

Alumnos:

- Fermin Alliot - ferminalliot@gmail.com
- Gabriel Antuña - gabrielantuna05@gmail.com

Materia: Programación I

Profesor: Cinthia Rigoni

Fecha de Entrega: 09/06/2025

Índice

1. Introducción

En el campo de la ciencia de la computación, la gestión eficiente de la información es un pilar fundamental. Los algoritmos de búsqueda y ordenamiento constituyen las herramientas esenciales que permiten a los desarrolladores organizar y acceder a los datos de manera rápida y precisa. La elección de un algoritmo sobre otro puede tener un impacto drástico en el rendimiento de una aplicación, especialmente cuando se manejan grandes volúmenes de datos.

Este trabajo tiene como objetivo principal investigar, analizar y demostrar de manera práctica el funcionamiento de los algoritmos de búsqueda y ordenamiento más representativos. Se busca no solo comprender su base teórica y su complejidad computacional (Big O), sino también observar su comportamiento en un entorno controlado y visual.

Para lograrlo, se ha desarrollado una aplicación web interactiva que sirve como un laboratorio virtual, permitiendo visualizar paso a paso la ejecución de cada algoritmo y comparar su rendimiento de forma empírica. A través de esta combinación de teoría

y práctica, aspiramos a consolidar una comprensión profunda sobre cuándo y por qué utilizar un algoritmo específico, una habilidad crucial en el desarrollo de software de calidad.

2. Marco Teórico

Algoritmos de Ordenamiento

Son procedimientos utilizados para reestructurar una colección de elementos (como una lista de números) siguiendo un criterio de orden específico (ascendente, descendente, etc.).

- **Bubble Sort (Ordenamiento de Burbuja):** Es uno de los algoritmos más sencillos. Compara repetidamente elementos adyacentes y los intercambia si están en el orden incorrecto. El proceso se repite hasta que la lista está ordenada. Su simplicidad tiene un costo en eficiencia, con una complejidad temporal de $\mathbf{O(n^2)}$ en el caso promedio y peor. Es útil con fines didácticos pero ineficiente para conjuntos de datos grandes.
- **Selection Sort (Ordenamiento por Selección):** Este método divide la lista en dos sublistas: una ordenada y otra desordenada. En cada iteración, encuentra el elemento más pequeño de la sublista desordenada y lo intercambia con el primer elemento de esta, moviéndolo así a la sublista ordenada. Al igual que Bubble Sort, su complejidad es $\mathbf{O(n^2)}$, haciéndolo poco práctico para grandes volúmenes de datos.
- **Insertion Sort (Ordenamiento por Inserción):** Construye la lista ordenada final un elemento a la vez. Recorre la lista y, para cada elemento, lo "inserta" en la posición correcta dentro de la parte ya ordenada. Su complejidad promedio es $\mathbf{O(n^2)}$, pero en el mejor de los casos (cuando la lista está casi ordenada) puede alcanzar $\mathbf{O(n)}$, lo que lo hace útil en ciertos escenarios específicos.
- **Quick Sort (Ordenamiento Rápido):** Es un algoritmo de tipo "divide y vencerás" y uno de los más eficientes y utilizados. Selecciona un elemento como "pivote" y particiona la lista en dos sub-listas: una con los elementos menores al pivote y otra con los mayores. Luego, aplica el mismo proceso recursivamente a las sub-listas. Su complejidad promedio es de $\mathbf{O(n \log n)}$, aunque en el peor caso (con un mal pivote, como en una lista ya ordenada) puede degradarse a $\mathbf{O(n^2)}$.
- **Merge Sort (Ordenamiento por Fusión):** Al igual que Quick Sort, es un algoritmo de tipo "divide y vencerás". Su estrategia consiste en dividir la lista desordenada recursivamente en sub-listas hasta que cada una contenga un solo elemento (lo cual se considera ordenado). Luego, combina (fusiona) estas sub-listas de manera ordenada para reconstruir gradualmente una lista final completamente ordenada. Su principal ventaja es que garantiza una complejidad

temporal de O(n log n) en todos los casos (mejor, promedio y peor). Su desventaja es que requiere espacio adicional (O(n)) para almacenar las sub-listas durante el proceso de fusión.

- **Counting Sort (Ordenamiento por Conteo):** Es un algoritmo de ordenamiento que no se basa en la comparación entre elementos. Funciona contando el número de ocurrencias de cada valor único dentro de un rango específico. Luego, utiliza estos conteos para calcular la posición de cada elemento en la secuencia de salida ordenada. Su gran ventaja es su eficiencia: tiene una complejidad temporal de O(n+k), donde n es el número de elementos y k es el rango de los valores de entrada. Esto lo hace extremadamente rápido (tiempo lineal) cuando k es pequeño y conocido. Sin embargo, su uso es limitado a números enteros y es impráctico si el rango de valores es muy grande, debido al requerimiento de memoria para el arreglo de conteo.

Algoritmos de Búsqueda

Su finalidad es localizar la posición de un elemento específico dentro de una estructura de datos.

- **Linear Search (Búsqueda Lineal):** Es el método más intuitivo. Recorre la colección de datos secuencialmente, elemento por elemento, desde el principio hasta el final, hasta encontrar el valor buscado o agotar la lista. No requiere que los datos estén ordenados. Su complejidad es **O(n)**, ya que en el peor de los casos debe revisar todos los elementos.
- **Binary Search (Búsqueda Binaria):** Es un algoritmo de búsqueda altamente eficiente, pero con un requisito indispensable: la colección de datos debe estar **previamente ordenada**. Funciona dividiendo repetidamente el intervalo de búsqueda por la mitad. Compara el valor buscado con el elemento del medio; si no es el valor, elimina la mitad en la que el valor no puede estar y continúa la búsqueda en la mitad restante. Este proceso reduce drásticamente el número de comparaciones, resultando en una complejidad de **O(log n)**.

3. Caso Práctico: Aplicación de Algoritmos de Ordenamiento a Problemas Específicos

En esta sección se presentan tres problemas concretos, cada uno diseñado para ilustrar la idoneidad de un algoritmo de ordenamiento específico. A través de la implementación en Python y el análisis de las decisiones de diseño, se busca demostrar cómo las características particulares de cada problema y los datos involucrados guían la elección del método de ordenamiento más eficiente.

3.1. Problema 1: Gestión de Lista de Tareas Pendientes con Prioridades

Dinámicas

3.1.1. Breve descripción del problema a resolver

Se requiere desarrollar un sistema simple para gestionar una lista de tareas pendientes. Cada tarea tiene una descripción y una prioridad asignada (un número entero donde 1 es la más alta prioridad). Frecuentemente, se añaden nuevas tareas a la lista. Después de cada modificación, la lista debe ser reordenada para que el usuario siempre visualice las tareas más urgentes primero. Se asume que la lista de tareas no es excesivamente larga (ej. menos de 50 tareas) y que, entre adiciones, la lista a menudo permanece **casi ordenada**.

3.1.2. Solución Propuesta: Insertion Sort (Ordenamiento por Inserción)

Para este escenario, se propone el algoritmo de **Ordenamiento por Inserción (Insertion Sort)**. Este algoritmo es particularmente adecuado para conjuntos de datos pequeños y para listas que ya están parcialmente ordenadas, condiciones que se alinean perfectamente con la descripción del problema.

3.1.3. Código fuente comentado

A continuación, se presenta el código Python que implementa la solución. Cada tarea se representa como un diccionario.

```
# Definición de la función insertion_sort para ordenar la lista de tareas

def insertion_sort_tareas(lista_tareas):

    # Recorremos la lista desde el segundo elemento (índice 1)

    # ya que el primer elemento se considera la sublista ordenada inicial.

    for i in range(1, len(lista_tareas)):

        # Guardamos el elemento actual que queremos insertar en su
posición correcta.

        tarea_actual = lista_tareas[i]

        # Guardamos la prioridad de la tarea actual para las
comparaciones.

        prioridad_actual = tarea_actual['prioridad']
```

```
        # Inicializamos j al índice del elemento anterior al actual.
        j = i - 1
        # Movemos los elementos de la sublista ordenada
(lista_tareas[0...i-1])
        # que tienen una prioridad menor (representada por un número
mayor)
        # que la prioridad_actual, una posición hacia adelante para hacer
        # espacio para la tarea_actual.
        # Se asume que un menor número de 'prioridad' significa mayor
urgencia.
        while j >= 0 and prioridad_actual < lista_tareas[j]['prioridad']:
            lista_tareas[j + 1] = lista_tareas[j]
            j -= 1
        # Insertamos la tarea_actual en su posición correcta dentro de la
sublista ordenada.
        lista_tareas[j + 1] = tarea_actual
    return lista_tareas
```

# --- Validación del Funcionamiento ---
# Lista inicial de tareas, ya ordenada.
tareas = [
  {'descripcion': 'Preparar informe mensual', 'prioridad': 2},
  {'descripcion': 'Enviar correos de seguimiento', 'prioridad': 3},
  {'descripcion': 'Planificar reunión de equipo', 'prioridad': 4}
]

Simulación de la adición de una nueva tarea urgente.
La lista ahora está "casi ordenada".

```
nueva_tarea = {'descripcion': 'Resolver bug crítico en producción', 'prioridad': 1}
tareas.append(nueva_tarea)

print("Lista de tareas ANTES de ordenar:")
for tarea in tareas:
    print(f"  - {tarea['descripcion']} (Prioridad: {tarea['prioridad']})")

# Aplicamos el ordenamiento por inserción
tareas_ordenadas = insertion_sort_tareas(tareas)

print("\nLista de tareas DESPUÉS de ordenar por prioridad:")
for tarea in tareas_ordenadas:
    print(f"  - {tarea['descripcion']} (Prioridad: {tarea['prioridad']})")
```

3.1.4. Explicación de decisiones de diseño

La elección de **Insertion Sort** se fundamenta en sus ventajas en este contexto específico:

- **Eficiencia en datos pequeños y casi ordenados**: Su principal ventaja. Insertion Sort tiene una complejidad de **O(n)** en el mejor de los casos (lista ya ordenada) y un rendimiento excelente para listas "casi ordenadas". En nuestro problema, añadir una nueva tarea solo perturba ligeramente el orden. Insertion Sort integrará eficientemente el nuevo elemento con un número mínimo de comparaciones y desplazamientos. Para listas pequeñas (< 50 elementos), su bajo costo computacional (overhead) a menudo lo hace más rápido que algoritmos teóricamente superiores como Quick Sort.
- **Bajo Overhead y Simplicidad**: Es un algoritmo simple de implementar que no requiere estructuras de datos adicionales ni llamadas recursivas, lo que lo hace muy eficiente en términos de memoria (O(1) espacio adicional) y factores constantes de ejecución.
- **Estabilidad**: Insertion Sort es un algoritmo estable, lo que significa que si dos tareas tuvieran la misma prioridad, su orden relativo se mantendría después del ordenamiento.

En comparación con otras opciones, **Quick Sort** tendría una sobrecarga innecesaria para una lista tan pequeña. **Bubble Sort** o **Selection Sort**, aunque simples, no aprovechan el estado "casi ordenado" de los datos y tendrían un rendimiento inferior, cercano a O(n2).

3.1.5. Validación del funcionamiento

Al ejecutar el código, se observa que la nueva_tarea con prioridad: 1 se añade al final de la lista. Tras aplicar insertion_sort_tareas, la salida muestra la lista reorganizada, con la tarea de prioridad 1 correctamente ubicada al principio, demostrando que el algoritmo funciona como se esperaba.

3.2. Problema 2: Procesamiento de un Gran Volumen de Registros de Ventas

3.2.1. Breve descripción del problema a resolver

Una empresa de comercio electrónico necesita generar un reporte que consolide todos los identificadores de transacción (IDs numéricos únicos) del último trimestre. Se trata de una lista con **millones de IDs** extraídos de diversas bases de datos, por lo que **no tienen un orden predefinido**. Para su procesamiento y análisis, es crucial ordenar esta gran masa de datos de la forma más rápida posible.

3.2.2. Solución Propuesta: Quick Sort (Ordenamiento Rápido)

Para ordenar un volumen masivo de datos desordenados, se propone el algoritmo **Quick Sort**. Es ampliamente reconocido por su eficiencia promedio de **O(n log n)** y su buen rendimiento en la práctica para grandes conjuntos de datos.

3.2.3. Código fuente comentado

El siguiente código implementa Quick Sort utilizando el esquema de partición de Lomuto.

```
import random # Para generar datos de prueba
```

Función de partición (utilizando el esquema de Lomuto)

```
def partition(arr, low, high):

    # Se elige el último elemento como pivote.

    # Mejoras comunes incluyen seleccionar un pivote aleatorio o la
mediana de tres

    # para mitigar el riesgo del peor caso.

    pivot = arr[high]

    i = low - 1  # 'i' es el índice del último elemento que es menor que
```

```
el pivote.
    # Se recorren los elementos desde 'low' hasta 'high-1'.
    for j in range(low, high):
        # Si el elemento actual es menor o igual al pivote.
        if arr[j] <= pivot:
            i += 1 # Se incrementa el índice 'i'.
            arr[i], arr[j] = arr[j], arr[i] # Se intercambia arr[i] con
arr[j].
                                        # Esto mueve los elementos
menores que el pivote a la izquierda.
    # Finalmente, se coloca el pivote en su posición correcta.
    # Se intercambia arr[i+1] (el primer elemento mayor que el pivote) con
arr[high] (el pivote).
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1 # Se retorna el índice donde el pivote ha sido colocado.


# Función principal de Quick Sort
def quick_sort(arr, low, high):
    if low < high: # Condición base: si hay más de un elemento para
ordenar.
        # 'pi' es el índice de partición. arr[pi] está ahora en su lugar
definitivo.
        pi = partition(arr, low, high)
        # Se ordenan recursivamente los elementos antes de la partición
```

```
        # y después de la partición.

        quick_sort(arr, low, pi - 1)

        quick_sort(arr, pi + 1, high)

    return arr # Retorna el arreglo ordenado (la modificación es
in-place).
```

# --- Validación del Funcionamiento ---
# Simulamos una lista grande de IDs de transacción. Para la demo, usamos 20.
# En un caso real, esto serían millones de elementos.
transaction_ids = [random.randint(1000000, 9999999) for _ in range(20)]

print(f"Lista de IDs antes de ordenar (muestra): \n{transaction_ids[:10]}...")

# Aplicamos Quick Sort
ids_ordenados = quick_sort(transaction_ids)

print(f"\nLista de IDs DESPUÉS de ordenar (muestra): \n{ids_ordenados[:10]}...")

3.2.4. Explicación de decisiones de diseño

Quick Sort es la elección preferida para este escenario por varias razones clave:

- **Eficiencia promedio** O(n log n): Para grandes volúmenes de datos desordenados, su rendimiento es excelente y, en la práctica, a menudo supera a otros algoritmos de la misma complejidad (como Merge Sort) debido a una mejor localidad de caché.
- **Ordenamiento "in-place"**: La implementación estándar de Quick Sort modifica la lista original directamente, requiriendo solo una cantidad de memoria auxiliar logarítmica (O(logn)) para la pila de recursión. Esto es una ventaja crucial frente a, por ejemplo, **Merge Sort**, que necesita O(n) espacio adicional, lo cual puede ser inviable con millones de registros.
- **Propósito general**: Es muy versátil y funciona bien con cualquier tipo de datos comparables.

La principal desventaja es su **peor caso de** O(n2), que puede ocurrir si se eligen malos pivotes consistentemente (por ejemplo, en una lista ya ordenada si se usa el primer/último elemento). Esto se mitiga fácilmente en implementaciones de

producción utilizando una estrategia de selección de pivote más robusta, como la "mediana de tres" o un pivote aleatorio.

Los algoritmos O(n2) como **Insertion Sort** serían extremadamente ineficientes e inviables para este volumen de datos.

3.2.5. Validación del funcionamiento

Al ejecutar el código de ejemplo, la lista desordenada de transaction_ids se procesa mediante quick_sort. La salida final muestra los mismos IDs, pero ahora dispuestos en orden numérico ascendente, validando que el algoritmo ha funcionado correctamente. Para una prueba completa, se verificaría con listas mucho más grandes y se medirían los tiempos de ejecución.

3.3. Problema 3: Clasificación de Resultados de Encuestas por Edad

3.3.1. Breve descripción del problema a resolver

Una organización necesita procesar los resultados de una encuesta a gran escala. Específicamente, debe ordenar una lista que contiene las edades de decenas de miles de participantes. Se sabe que las edades son **números enteros** y caen dentro de un **rango demográfico limitado y conocido** (por ejemplo, entre 18 y 99 años). El objetivo es ordenar esta lista de la manera más eficiente posible.

3.3.2. Solución Propuesta: Counting Sort (Ordenamiento por Conteo)

Para un problema donde los datos son enteros dentro de un rango (k) conocido y relativamente pequeño en comparación con el número de elementos (n), **Counting Sort** es la solución óptima. Es un algoritmo de ordenamiento no comparativo que alcanza una eficiencia de **tiempo lineal**.

3.3.3. Código fuente comentado

El siguiente código implementa Counting Sort aprovechando el rango conocido de las edades.

```
# Función Counting Sort para ordenar edades dentro de un rango conocido

def counting_sort_edades(lista_edades, min_edad, max_edad):

    # El tamaño del array de conteo (count_array) se determina por el
rango de edades.
```

```
    # rango_edades = max_edad - min_edad + 1

    # count_array[i] almacenará el número de ocurrencias de la edad
(min_edad + i).

    count_array = [0] * (max_edad - min_edad + 1)

    # El array de salida (output_array) tendrá el mismo tamaño que la
lista original.

    output_array = [0] * len(lista_edades)



    # Paso 1: Contar la frecuencia de cada edad.

    # Se itera sobre la lista de edades original.

    for edad in lista_edades:

        # Se verifica que la edad esté dentro del rango esperado.

        if min_edad <= edad <= max_edad:

            # Se incrementa el contador para la edad correspondiente.

            # Se ajusta la edad al índice del count_array restando
min_edad.

            count_array[edad - min_edad] += 1

        # else: Aquí se podría manejar edades fuera de rango (e.g.,
ignorar, registrar error).



    # Paso 2: Modificar el count_array para que cada índice

    # almacene la posición acumulada. Es decir, count_array[i] contendrá

    # la suma de las frecuencias de las edades hasta (min_edad + i).

    # Esto indica la posición final (basada en 1) de los elementos con esa
```

```
edad en el output_array.

    for i in range(1, len(count_array)):

        count_array[i] += count_array[i - 1]



    # Paso 3: Construir el array de salida (output_array).

    # Se itera la lista_edades original en orden inverso. Esto es
importante

    # para mantener la estabilidad del algoritmo (si hubiera datos
asociados a las edades).

    for i in range(len(lista_edades) - 1, -1, -1):

        edad_actual = lista_edades[i]

        if min_edad <= edad_actual <= max_edad:

            # La posición en output_array se determina por
count_array[edad_actual - min_edad] - 1

            # (se resta 1 porque las posiciones del array son base 0).

            output_array[count_array[edad_actual - min_edad] - 1] =
edad_actual

            # Se decrementa el conteo para la edad actual, para colocar
correctamente

            # la próxima ocurrencia de la misma edad (si existe) en la
posición anterior.

            count_array[edad_actual - min_edad] -= 1

    return output_array
```

--- Validación del Funcionamiento ---

```
# Lista de ejemplo con edades. Rango conocido: 18 a 99.
edades = [25, 42, 18, 55, 42, 22, 19, 38, 25, 18, 60]
min_edad_rango = 18
max_edad_rango = 99

print(f"Lista de edades antes de ordenar: \n{edades}")

# Aplicamos Counting Sort
edades_ordenadas = counting_sort_edades(edades, min_edad_rango,
max_edad_rango)

print(f"\nLista de edades DESPUÉS de ordenar: \n{edades_ordenadas}")
```

3.3.4. Explicación de decisiones de diseño

Counting Sort es la opción ideal aquí por una razón fundamental:

- **Complejidad de tiempo lineal** O(n+k): Donde n es el número de encuestados y k es el rango de edades (99−18+1=82). Como k es una constante pequeña, la complejidad efectiva es **O(n)**. Esto es asintóticamente más rápido que cualquier algoritmo de ordenamiento basado en comparaciones (como Quick Sort o Merge Sort), que no pueden superar O(n log n).
- **No basado en comparaciones**: El algoritmo no compara elementos entre sí. En su lugar, utiliza los valores de las edades como índices en un array auxiliar para contar sus ocurrencias. Este enfoque es lo que le permite romper la barrera de O(n log n).

La principal **restricción** de Counting Sort es que solo es práctico si el rango k no es excesivamente grande, ya que requiere un array auxiliar de tamaño k, lo que implica un uso de memoria de O(k). Para este problema, un array de 82 elementos es trivial. Sin embargo, sería inviable para ordenar números de 32 bits, donde k sería de miles de millones.

Quick Sort (O (n log n)) sería la segunda mejor opción, pero notablemente más lento que la solución de tiempo lineal que las propiedades de los datos nos permiten usar.

3.3.5. Validación del funcionamiento

El código de ejemplo toma una lista de edades desordenada, incluyendo duplicados (18, 25, 42) y valores de los extremos del rango. La salida impresa muestra la lista perfectamente ordenada de menor a mayor, confirmando que el algoritmo maneja

correctamente los datos y sus repeticiones.

4. Metodología Utilizada

El proyecto se abordó siguiendo un proceso estructurado en varias fases:

1. **Investigación y Fundamentación:** Se realizó una revisión exhaustiva de la bibliografía recomendada, documentación oficial y recursos en línea (como Visualgo.net) para consolidar el marco teórico de cada algoritmo.
2. **Planificación y Diseño:** Se definió el alcance del caso práctico y se diseñó la arquitectura de la aplicación web, enfocándose en una interfaz de usuario clara, intuitiva y educativa.
3. **Desarrollo e Implementación:** Se procedió a la codificación de la aplicación. La lógica de cada algoritmo fue implementada en Python.
4. **Pruebas y Validación:** La aplicación se sometió a pruebas rigurosas con diferentes tipos y tamaños de datos para asegurar que los algoritmos funcionaran correctamente y que las visualizaciones fueran precisas. Se validó que el contador de operaciones y el comparador de rendimiento reflejaran los resultados esperados.
5. **Análisis de Resultados:** Se utilizaron las herramientas de la propia aplicación para recopilar datos de rendimiento y se analizaron las conclusiones extraídas de la visualización.
6. **Elaboración del Informe:** Finalmente, se redactó este documento para consolidar toda la investigación, el desarrollo práctico y los hallazgos del proyecto.

5. Resultados Obtenidos

El desarrollo del "Explorador Interactivo de Algoritmos" permitió obtener los siguientes resultados:

- **Validación Empírica de la Complejidad:** La herramienta de comparación de rendimiento demostró de forma clara el impacto de la complejidad Big O. Al probar con N=10,000 elementos aleatorios, los algoritmos $O(n^2)$ (Bubble, Selection, Insertion) tardaron varios segundos, mientras que Quick Sort ($O(n \log n)$) completó la tarea en milisegundos.
- **Comprensión Profunda del Funcionamiento:** La visualización paso a paso permitió identificar los puntos fuertes y débiles de cada algoritmo. Por ejemplo, se observó claramente cómo Insertion Sort es eficiente con datos casi ordenados y cómo la elección de un mal pivote puede degradar el rendimiento de Quick Sort.
- **Clarificación del Requisito de la Búsqueda Binaria:** La aplicación hace evidente que la Búsqueda Binaria falla o es inaplicable si la lista no está ordenada,

reforzando la sinergia entre los algoritmos de ordenamiento y búsqueda.
- **Herramienta Educativa Funcional:** El resultado final es una aplicación robusta que cumple su objetivo de servir como un recurso didáctico para explicar conceptos abstractos de una manera concreta y atractiva.

6. Conclusiones

Este trabajo integrador ha permitido no solo repasar, sino internalizar los conceptos fundamentales de los algoritmos de búsqueda y ordenamiento. La construcción de una herramienta desde cero para visualizarlos nos obligó a entender su lógica interna a un nivel de detalle que la simple lectura teórica no proporciona.

Hemos aprendido que:

1. **No existe un "mejor algoritmo" universal.** La elección correcta depende críticamente del tamaño, el estado inicial (ordenado, aleatorio) y la naturaleza de los datos, así como de los requisitos de rendimiento de la aplicación.
2. **La complejidad computacional (Big O) es una guía práctica y no un mero concepto abstracto.** Sus implicaciones en el mundo real son directas y medibles, como se demostró en el comparador de rendimiento.
3. **La visualización es una poderosa herramienta de aprendizaje en la programación.** Transforma líneas de código y diagramas de flujo en procesos animados y comprensibles, facilitando la identificación de patrones y la depuración de la lógica.

Como posibles mejoras futuras, se podría extender la aplicación para incluir otros algoritmos (Merge Sort, Heap Sort), estructuras de datos más complejas (árboles) y permitir al usuario introducir sus propios conjuntos de datos.

7. Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Google. (s.f.). *Colaboratory: Python, Algoritmos y Estructuras de Datos*. Recuperado de https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPreD9J
- Python Software Foundation. (2024). *Python 3 Documentation*. Recuperado de https://docs.python.org/3/
- VisuAlgo. (s.f.). *Visualising data structures and algorithms through animation*. Recuperado de https://visualgo.net/en

8. Anexos

- Aplicación Web - Explorador Interactivo de Algoritmos:

El caso práctico desarrollado es una aplicación web funcional. Se puede acceder a una demostración del mismo a través del video explicativo.

- Repositorio en GitHub:
  El código fuente completo, funcional y documentado se encuentra disponible en: https://github.com/Falliot00/UTN-TUPaD-P1/tree/main/Trabajo%20Integrador
- Video Explicativo:
  Se ha preparado un video tutorial que presenta la investigación, demuestra la aplicación desarrollada y reflexiona sobre los aprendizajes obtenidos: https://youtu.be/IobTp8QF_xo