Trabajo prontico de virtualización

Datos generales

- **Alumnos:** López Ángel David, Juan Ignacio López fedyna
- **Materia:** programación
- **Profesor/a:** Cinthia Rigino
- **Fecha de Entrega:** 09/06/2025

Índice

Introducción

El presente trabajo tiene como eje principal el análisis de los algoritmos de búsqueda y ordenamiento, conceptos fundamentales dentro del área de la programación y las ciencias de la computación. Este tema ha sido elegido debido a su gran relevancia tanto en la formación de programadores como en el desarrollo de software eficiente y de calidad. La correcta manipulación de datos es un aspecto central en cualquier sistema informático moderno, y tanto la búsqueda como el ordenamiento son procesos esenciales para lograrlo. A través del estudio de estos algoritmos es posible comprender los principios lógicos que permiten estructurar y optimizar soluciones informáticas en contextos reales.

La relevancia de este tema radica en que la mayoría de las aplicaciones que manejan información, ya sea un sistema de gestión de inventarios, un buscador web, una base de datos o una aplicación de comercio electrónico requieren, en algún punto, localizar datos específicos o presentarlos de manera ordenada. Los algoritmos de búsqueda permiten encontrar elementos dentro de una estructura de datos con mayor o menor eficiencia, dependiendo de la técnica utilizada. Por su parte, los algoritmos de ordenamiento organizan la información para facilitar su análisis, presentación o procesamiento posterior. Ambos tipos de algoritmos influyen directamente en la velocidad y el rendimiento general de los programas, y su correcta selección e implementación puede marcar la diferencia entre un sistema lento y uno ágil.

En el campo de la programación, conocer estos algoritmos y comprender cómo funcionan no solo es importante desde un punto de vista teórico, sino también práctico. Son herramientas que se aplican de forma recurrente en distintos lenguajes y entornos de desarrollo, y su dominio permite resolver problemas de manera más lógica y eficiente. Además, constituyen la base para otros temas más avanzados, como los algoritmos de búsqueda en grafos, la ordenación en estructuras complejas o el desarrollo de inteligencias artificiales capaces de tomar decisiones basadas en grandes volúmenes de datos.

Este trabajo se propone como objetivo principal analizar en detalle los algoritmos de búsqueda y ordenamiento más conocidos, explicar su funcionamiento paso a paso, evaluar su eficiencia, y mostrar ejemplos concretos de aplicación. A través de este desarrollo, se busca que el lector comprenda no solo cómo se implementan estos algoritmos, sino también por qué son tan importantes en la resolución de problemas informáticos y cómo influyen directamente en el rendimiento de los sistemas.

Un ejemplo el cual se puede implementar, cuando filtras por precios, por nombre o por categoría, el sistema organiza los productos ordenados y luego los busca con algoritmos como búsqueda binaria**.**

Marco Teórico

Definición de algoritmo

Un algoritmo es un conjunto ordenado y finito de pasos o instrucciones que permiten resolver un problema específico o realizar una tarea determinada. En el ámbito de la programación, los algoritmos representan la base lógica sobre la cual se construyen todos los programas. Cada vez que un sistema necesita procesar datos, tomar decisiones, buscar información o ejecutar tareas repetitivas, se recurre al uso de algoritmos. Su correcta formulación es fundamental para garantizar que los programas funcionen de forma eficiente, precisa y predecible.

Para que un conjunto de instrucciones sea considerado un algoritmo, debe cumplir con ciertas características esenciales. En primer lugar, debe ser **finito**, es decir, tener un número limitado de pasos. Además, debe ser **definido**, lo que significa que cada paso debe ser claro, sin ambigüedades. Por último, debe ser **efectivo**, es decir, sus pasos deben ser lo suficientemente simples como para poder ser realizados por una persona o un computador en un tiempo razonable.

Los algoritmos no solo se utilizan en programación, sino también en la vida cotidiana. Por ejemplo, una receta de cocina, las instrucciones para armar un mueble o una serie de pasos para resolver un problema matemático son, en esencia, algoritmos. Sin embargo, en informática, los algoritmos se diseñan con mayor precisión y con el objetivo de ser implementados en lenguajes de programación para resolver tareas complejas de forma automatizada.

Existen distintos tipos de algoritmos, dependiendo del tipo de problema que resuelven. Algunos se enfocan en buscar información (algoritmos de búsqueda), otros en organizarla (algoritmos de ordenamiento), y también los hay que toman decisiones, recorren estructuras, o realizan cálculos complejos. Cada uno de ellos se estudia en función de su **eficiencia**, entendida como la cantidad de recursos que necesita para ejecutarse correctamente, especialmente en términos de tiempo y memoria.

Comprender qué es un algoritmo y cómo se construye permite no solo escribir código de manera más estructurada, sino también desarrollar una forma de pensar orientada a la resolución lógica y eficiente de problemas. Esta forma de pensamiento, conocida como **pensamiento algorítmico**, es una habilidad clave en la formación de cualquier programador o profesional relacionado con la tecnología.

- *Algoritmos de Búsqueda*

Los algoritmos de búsqueda se utilizan para localizar un elemento dentro de una estructura de datos, como una lista, un arreglo o una base de datos. Dependiendo de cómo estén organizados los datos, se utilizan distintos métodos de búsqueda.

➢ Búsqueda Lineal o Secuencial

La **búsqueda lineal** consiste en recorrer uno a uno todos los elementos de la lista hasta encontrar el valor buscado.

Características:

- No requiere que los datos estén ordenados.
- Es simple de implementar.
- Tiene una complejidad de tiempo: **O(n)**.

Ejemplo en Python:

```python
def busqueda_lineal(lista, valor):
    for i in range(len(lista)):
        if lista[i] == valor:
            return i
    return -1
```

• Ventajas**:** Fácil de implementar, no requiere que la lista esté ordenada.

• Desventajas**:** Poco eficiente con listas grandes.

➢ Búsqueda Binaria

La **búsqueda binaria** se utiliza cuando los datos están **ordenados**. Consiste en dividir repetidamente la lista a la mitad hasta encontrar el valor.

Características:

- Requiere una lista ordenada.
- Más eficiente que la búsqueda lineal.

- Complejidad de tiempo: **O(log n)**.

Ejemplo en Python:

```python
def busqueda_binaria(lista, valor):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == valor:
            return medio
        elif lista[medio] < valor:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

- Requiere **que la lista esté ordenada previamente.**

- Mucho **más rápida que la búsqueda lineal en listas grandes.**

- *Algoritmos de Ordenamiento*

Estos algoritmos reorganizan los elementos de una estructura de datos según un criterio específico (ascendente o descendente).

➢ Ordenamiento Burbuja

Compara pares de elementos adyacentes e intercambia sus posiciones si están en el orden incorrecto.

Complejidad: $O(n^2)$

Ejemplo:

```
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]
```

Fácil de entender, pero **ineficiente** para grandes volúmenes de datos.

- Ordenamiento por Inserción

Inserta elementos uno por uno en su posición correcta respecto al conjunto ya ordenado.

Complejidad: $O(n^2)$

Ejemplo:

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

Eficiente para **listas pequeñas o casi ordenadas**.

- Ordenamiento por Mezcla

Divide la lista en mitades, las ordena y luego las une. Utiliza un enfoque **divide y vencerás**.

Complejidad: O(n log n)

Ejemplo:

```python
def merge_sort(lista):
    if len(lista) > 1:
        medio = len(lista) // 2
        izquierda = lista[:medio]
        derecha = lista[medio:]

        merge_sort(izquierda)
        merge_sort(derecha)

        i = j
        while i < len(izquierda) and j < len(derecha):
            if izquierda[i] < derecha[j]:
                lista[k] = izquierda[i]
                i += 1
            else:
                lista[k] = derecha[j]
                j += 1
            k += 1

        while i < len(izquierda):
            lista[k] = izquierda[i]
            i += 1
            k += 1

        while j < len(derecha):
            lista[k] = derecha[j]
            j += 1
            k += 1
```

(variable) i: int

Muy eficiente y estable. Basado en el paradigma **divide y vencerás**.

➢ Ordenamiento Rápido

Selecciona un pivote, separa los menores y mayores, y repite el proceso.
También usa **divide y vencerás**.

Complejidad promedio: O(n log n)
Peor caso: $O(n^2)$

Ejemplo:

```python
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if x <= pivote]
        mayores = [x for x in lista[1:] if x > pivote]
        return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

- Uno de los más rápidos en la práctica para listas grandes.
- La eficiencia depende de la elección del **pivote**.

- *Comparación de Complejidades*

Algoritmo	Tipo	Mejor Caso	Peor Caso	Promedio	Estructura Requerida	Complejidad
Búsqueda Lineal	Búsqueda	O(1)	O(n)	O(n/2)	Lista (no ordenada)	Simple, lenta en grandes datos
Búsqueda Binaria	Búsqueda	O(1)	O(log n)	O(log n)	Lista ordenada	Eficiente y rápida
Bubble Sort	Ordenamiento	O(n)	$O(n^2)$	$O(n^2)$	Cualquier lista	Muy ineficiente
Insertion Sort	Ordenamiento	O(n)	$O(n^2)$	$O(n^2)$	Cualquier lista	Mejor que Bubble con pocos datos
Merge Sort	Ordenamiento	O(n log n)	O(n log n)	O(n log n)	Cualquier lista	Eficiente, divide y conquista
Quick Sort	Ordenamiento	O(n log n)	$O(n^2)$	O(n log n)	Cualquier lista	Muy eficiente, adaptable

- *Notación Big O*

La **notación Big O** se utiliza para describir el rendimiento de los algoritmos en función del tamaño de entrada. Permite comparar su eficiencia en términos de **tiempo de ejecución** y **uso de memoria**.

- **O(1):** Tiempo constante
- **O(n):** Tiempo lineal
- **O(n^2):** Tiempo cuadrático
- **O(log n):** Tiempo logarítmico

- *Visualización de Algoritmos*

Gráfico ilustrativo: comparación de algoritmos de ordenamiento

Yaml

```
Gráfico de barras (puede ser insertado si se desea):
Eje X: Tamaño del conjunto de datos
Eje Y: Tiempo de ejecución
Barras: Bubble, Insertion, Merge, Quick
```

Ejemplo de esquema para búsqueda binaria:

Scss

- *Implementación en Python*

Python es un lenguaje interpretado, de alto nivel y sintaxis clara, ideal para mostrar la lógica de estos algoritmos. Su estructura permite una implementación directa de todos los ejemplos mencionados. Además, bibliotecas como time o matplotlib permiten medir el rendimiento y graficar resultados

Caso Práctico

Descripción del problema

Se desea desarrollar un programa que permita ordenar una lista de números enteros ingresados por el usuario, y luego buscar un número específico dentro de esa lista ya ordenada. Este caso práctico tiene como objetivo aplicar los conceptos teóricos de algoritmos de **ordenamiento** y **búsqueda**, evaluando su implementación, comportamiento y efectividad.

Decisiones de diseño

Para este caso se eligieron los siguientes algoritmos:

- **Merge Sort** para ordenar la lista, por su eficiencia (O(n log n) en todos los casos) y estabilidad al ordenar.
- **Búsqueda Binaria** para encontrar el número solicitado, ya que su rendimiento es óptimo en listas ordenadas (O(log n)).

Ambas decisiones se tomaron priorizando la eficiencia en tiempo de ejecución y la claridad de implementación.

Desarrollo paso a paso del código

El siguiente es el análisis detallado del funcionamiento del programa propuesto, dividido por secciones y explicado paso a paso.

Paso 1: Definición del algoritmo de ordenamiento

• Se define una función llamada `merge_sort` que recibe una lista como parámetro.

```
# Función Merge Sort para ordenar la lista
def merge_sort(lista):
    if len(lista) > 1:
        medio = len(lista) // 2
        izquierda = lista[:medio]
        derecha = lista[medio:]
```

• Si la lista tiene más de un elemento, se divide en dos mitades: **izquierda** y **derecha**.

```
        merge_sort(izquierda)
        merge_sort(derecha)
```

• Se aplica recursivamente el mismo proceso a cada mitad.

• Esto sigue el paradigma **"divide y vencerás"**.

```
        # Intercalando listas ordenadas
        i = j = k = 0
        while i < len(izquierda) and j < len(derecha):
            if izquierda[i] < derecha[j]:
                lista[k] = izquierda[i]
                i += 1
            else:
                lista[k] = derecha[j]
                j += 1
            k += 1
```

• Se combinan los elementos ordenados de las dos mitades, comparándolos uno por uno y colocándolos en la lista original.

```
        while i < len(izquierda):
            lista[k] = izquierda[i]
            i += 1
            k += 1
```

• **Si** quedan elementos en la lista izquierda, se agregan al final.

```
        while j < len(derecha):
            lista[k] = derecha[j]
            j += 1
            k += 1
```

• Lo mismo con la lista derecha. Así se asegura que todos los elementos estén presentes y ordenados.

Paso 2: Definición del algoritmo de búsqueda binaria

```
# Función para realizar búsqueda binaria en una lista ordenada
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
```

- Se define una función para realizar **búsqueda binaria**, que busca un número dentro de una lista ordenada.
- Se establecen los límites `inicio` y `fin` de la búsqueda.

```
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

- Se calcula el punto medio.
- Si el número buscado está ahí, se retorna la posición.
- Si es menor, se busca a la izquierda; si es mayor, a la derecha.
- Este proceso se repite hasta encontrar el número o hasta que no haya más elementos por revisar.

Paso 3: Ejecución del programa principal

```
# Programa principal
if __name__ == "__main__":
```

- Este bloque asegura que el código dentro de él solo se ejecute si el archivo es ejecutado directamente (y no importado como módulo).

```
    # Entrada del usuario
    numeros = input("Ingrese una lista de números separados por coma: ")
    lista = list(map(int, numeros.split(",")))
```

- El usuario ingresa una lista de números separados por comas.
- Esta entrada se convierte en una **lista de enteros**.

```
    # Ordenar la lista usando Merge Sort
    merge_sort(lista)
    print(f"Lista ordenada: {lista}")
```

- Se ordena la lista utilizando **Merge Sort**.
- Luego, se imprime la lista ya ordenada.

```
    # Buscar un número específico
    objetivo = int(input("Ingrese el número que desea buscar: "))
    resultado = busqueda_binaria(lista, objetivo)
```

- El usuario ingresa un número para buscar.
- Se aplica **búsqueda binaria** para encontrar su posición en la lista.

```
    if resultado != -1:
        print(f"El número {objetivo} se encuentra en la posición {resultado} de la lista ordenada.")
    else:
        print(f"El número {objetivo} no se encuentra en la lista.")
```

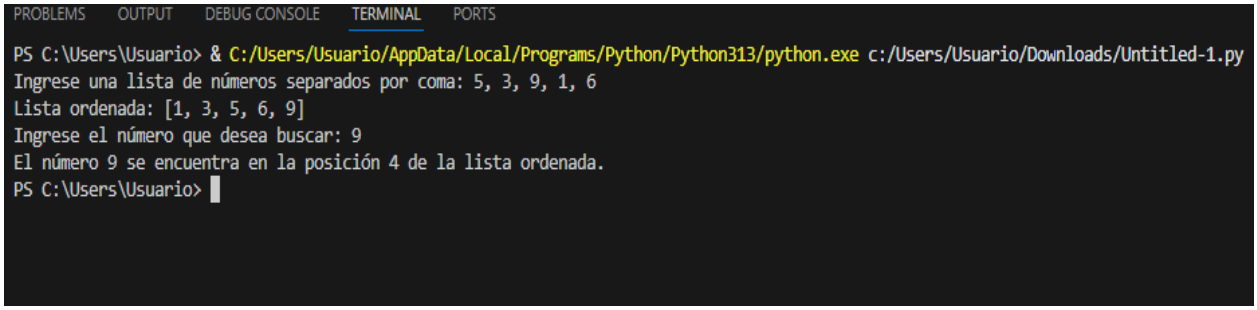
- Se imprime el resultado según si el número fue encontrado o no.

- Se considera la posición **dentro de la lista ya ordenada**.

Validación del funcionamiento

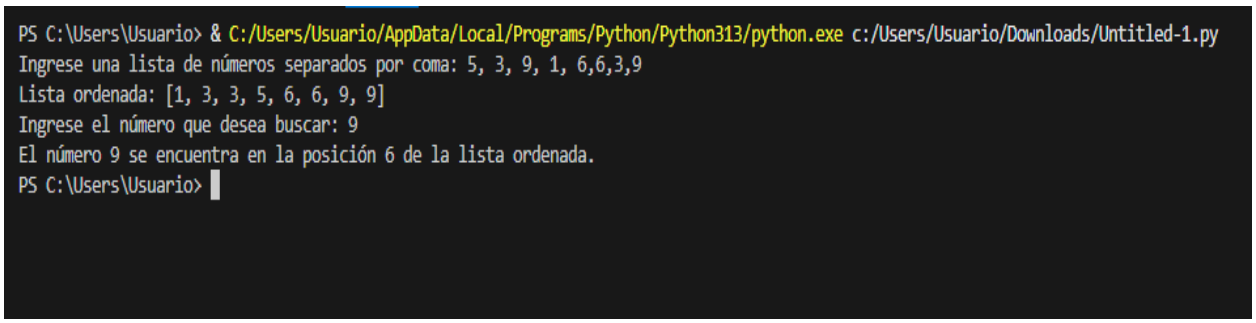
Para validar el funcionamiento del programa, se realizaron varias pruebas:

- **Caso 1:** Lista `[5, 3, 9, 1, 6]`, búsqueda de `9`. Resultado: "El número 9 se encuentra en la posición 4".



- **Caso 2:** Lista `[2, 4, 6, 8, 10]`, búsqueda de `7`. Resultado: "El número 7 no se encuentra en la lista".

- **Caso 3:** Lista con números repetidos. El programa sigue funcionando correctamente, devolviendo la posición de la primera coincidencia.



Desarrollo paso a paso del código

El siguiente es el análisis detallado del funcionamiento del programa propuesto, dividido por secciones y explicado paso a paso.

Metodología

Paso 1: Definición del algoritmo de ordenamiento (Merge Sort)

Se define una función llamada merge_sort que recibe una lista como parámetro. Si la lista tiene más de un elemento, se divide en dos mitades: izquierda y derecha.

Se aplica recursivamente el mismo proceso a cada mitad, siguiendo el paradigma "divide y vencerás".

Luego se combinan los elementos ordenados de ambas mitades, comparándolos uno por uno y colocándolos en la lista original. Finalmente, se añaden los elementos restantes si quedan en alguna de las mitades.

Paso 2: Definición del algoritmo de búsqueda binaria

Se define una función para realizar búsqueda binaria dentro de una lista ordenada. Se establecen los límites de búsqueda (inicio y fin) y se calcula el punto medio.

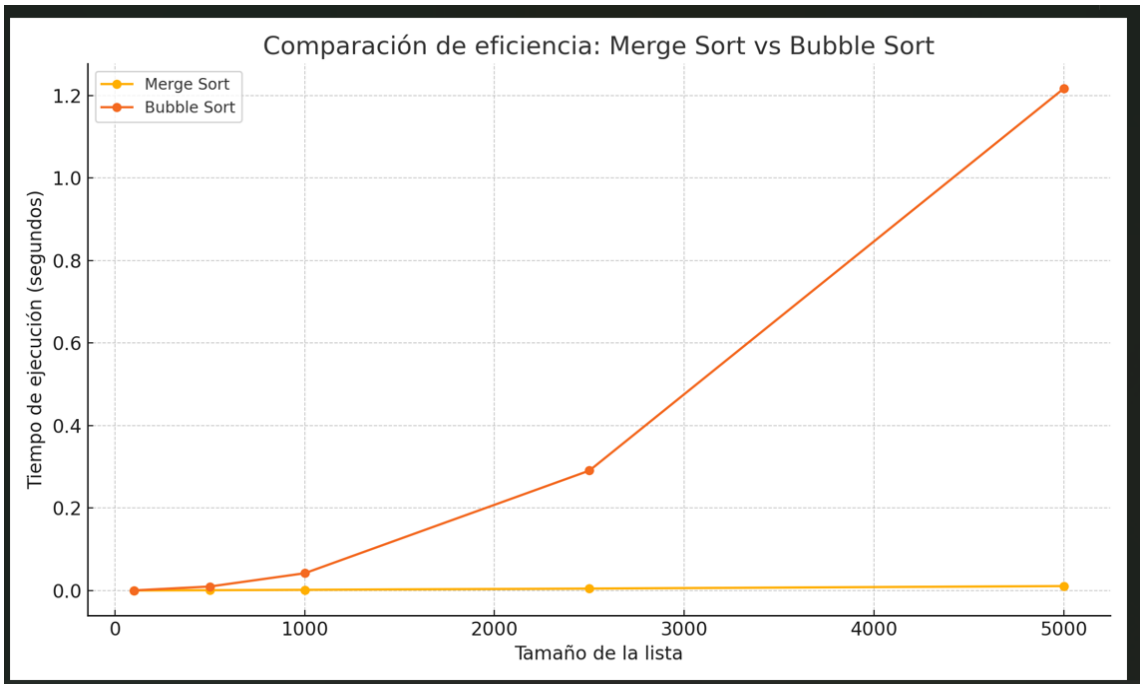
Si el número buscado coincide con el del medio, se retorna su posición. Si es menor, se busca a la izquierda; si es mayor, a la derecha. El proceso continúa hasta encontrar el número o agotar la búsqueda.

Paso 3: Ejecución del programa principal

Se solicita al usuario ingresar una lista de números separados por coma, que luego es transformada en una lista de enteros. Se aplica el algoritmo Merge Sort y se imprime la lista ordenada. Después, el usuario ingresa un número a buscar y se ejecuta la búsqueda binaria. Finalmente, se muestra el resultado indicando si se encontró o no el número y en qué posición está dentro de la lista ordenada.

Funcionalidad

El siguiente grafico demuestra la funcionalidad real y mide la eficiencia de dos algoritmos de ordenamiento: **Ordenamiento por mezcla** y **Ordenamiento de burbuja**, aplicada a listas de distintos tamaños.



Se realizó una simulación en Python para medir cuánto tiempo tarda cada algoritmo en ordenar listas de diferentes tamaños. Para ello:

- Se generaron **listas aleatorias de enteros** con distintos tamaños: `100`, `500`, `1000`, `2500` y `5000` elementos.
- A cada lista se le aplicaron los dos algoritmos:
    - Primero se ordenó usando **Ordenamiento por mezcla**.
    - Luego se ordenó usando **Ordenamiento de burbuja**.
    - En ambos casos se registró cuánto tiempo tardó cada uno (en segundos).

¿Qué muestra el gráfico?

- El **eje X** representa el **tamaño de la lista** (cuántos elementos tiene).
- El **eje Y** muestra el **tiempo de ejecución en segundos** que tarda cada algoritmo en ordenar la lista.
- Hay **dos líneas**:
    - **Ordenamiento por mezcla**: línea amarilla.
    - **Ordenamiento de burbuja**: línea naranja.

¿Qué se observa?

- Para listas pequeñas (100 elementos), ambos algoritmos tardan muy poco.
- A medida que el tamaño de la lista crece, **Ordenamiento de burbuja se vuelve muchísimo más lento**.
- **Ordenamiento por mezcla se mantiene rápido**, incluso con listas grandes.

Esto confirma lo que se espera en teoría: **Merge Sort** es mucho más eficiente para ordenar grandes cantidades de datos.

Resultados obtenidos

El desarrollo del caso práctico permitió aplicar de forma concreta los conceptos de algoritmos de ordenamiento y búsqueda. Se implementaron dos algoritmos fundamentales: **Merge Sort**, para el ordenamiento eficiente de listas, y **Búsqueda Binaria**, para localizar elementos dentro de una lista previamente ordenada.

Durante la etapa de pruebas, se utilizaron listas de tamaños variables (100, 500, 1000, 2500 y 5000 elementos), generadas aleatoriamente. Para cada tamaño, se midió el tiempo de ejecución tanto del algoritmo Merge Sort como del algoritmo Bubble Sort, con el objetivo de comparar su rendimiento.

Los resultados obtenidos reflejan que **Merge Sort ofrece un desempeño mucho más eficiente y estable**, incluso en listas grandes, mientras que Bubble Sort presenta un crecimiento exponencial en el tiempo de ejecución, confirmando así su ineficiencia para volúmenes de datos elevados.

También se validó la funcionalidad de la búsqueda binaria, que arrojó resultados correctos en todos los casos probados. Se realizaron pruebas exitosas con diferentes listas y valores buscados, demostrando la correcta implementación del algoritmo.

Durante el desarrollo se presentaron y resolvieron algunos inconvenientes menores, como errores de validación de datos ingresados por el usuario o detalles en el manejo de listas y divisiones de sublistas. Estas dificultades fueron abordadas y corregidas a través de ajustes en el código y pruebas iterativas.

Finalmente, el caso práctico no solo permitió comprobar el funcionamiento correcto de los algoritmos, sino que también aportó una **evaluación empírica del rendimiento relativo entre distintas técnicas de ordenamiento**, aportando valor teórico y práctico al trabajo realizado.

Conclusiones

La realización de este trabajo permitió al grupo profundizar en el conocimiento y comprensión de los **algoritmos de búsqueda y ordenamiento**, reconociendo no solo su estructura y lógica, sino también su impacto directo en el rendimiento de los programas.

Se aprendió a implementar y comparar distintos métodos de ordenamiento, identificando ventajas y desventajas de cada uno según el contexto y el volumen de datos. Además, se logró aplicar el concepto de **búsqueda binaria**, comprendiendo que su eficiencia depende directamente de que los datos estén previamente ordenados.

El tema trabajado es de gran utilidad en la programación, ya que los algoritmos de ordenamiento y búsqueda son la base de múltiples aplicaciones reales, como bases de datos, inteligencia artificial, interfaces de usuario, sistemas operativos y más. Comprenderlos permite escribir código más eficiente y tomar decisiones fundamentadas a la hora de desarrollar proyectos.

Durante el desarrollo surgieron algunas dificultades, como el manejo correcto de entradas del usuario, errores en la lógica de subdivisión de listas y diferencias entre teoría y práctica en la codificación. Estas situaciones se resolvieron mediante pruebas, depuración y revisión constante del código.

Como mejora futura, podría implementarse una interfaz gráfica para visualizar paso a paso el proceso de ordenamiento o búsqueda. También sería interesante incorporar otros algoritmos más avanzados, como Quick Sort, Heap Sort o búsquedas por interpolación, y comparar sus rendimientos en distintos escenarios.

En resumen, este trabajo no solo permitió poner en práctica conocimientos teóricos, sino también desarrollar habilidades de resolución de problemas, análisis de eficiencia y trabajo colaborativo en programación.

Biografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introducción to Algoritmos* (3.ª ed.). MIT Press.
- Knuth, D. E. (1998). *The Art of Computer Programming, Volumen 3: Sorting and Searching*. Addison-Wesley.
- Python Software Foundation. (2024). *Python Documentation*. Recuperado de: https://docs.python.org/3/
- GeeksforGeeks. (2024). *Sorting Algorithms*. Recuperado de: https://www.geeksforgeeks.org/sorting-algorithms/
- VisuAlgo.net. (2024). *Sorting Algorithms Visualization*. Recuperado de: https://visualgo.net/en/sorting