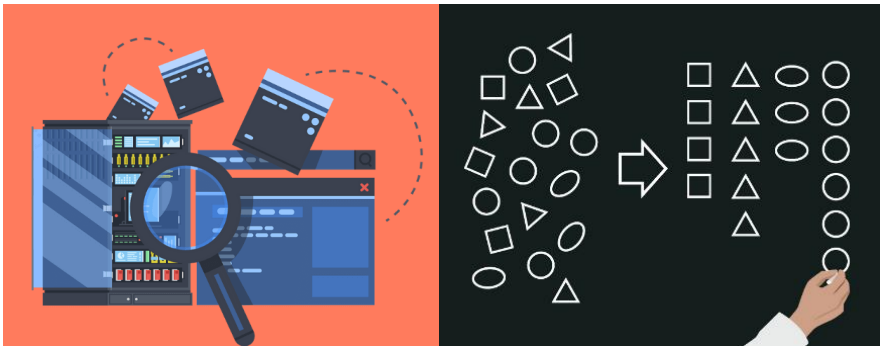
Universidad Tecnológica Nacional

Tecnicatura Universitaria en Programación a Distancia

Programación I

TRABAJO INTEGRADOR

ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO EN PYTHON



Integrantes:

- Arjona, Martín Joel - *87martin@hotmail.es*
- Arjona, Paola Yanina del Valle - *pao.arjo2014@gmail.com*

Profesor: Bruselario, Sebastián

Comisión: N°24

09/06/2025

INDICE

INTRODUCCIÓN

En programación, los algoritmos de búsqueda y ordenamiento son herramientas fundamentales, ya que permiten organizar y acceder a grandes volúmenes de datos de forma eficiente. Su correcta implementación impacta directamente en el rendimiento de los sistemas, por lo que comprender cómo funcionan y en qué situaciones conviene usar uno u otro es importante para diseñar soluciones optimizadas.

Por otro lado, en la era de la transformación digital, la gestión eficiente de la información es un elemento esencial para garantizar el funcionamiento óptimo de cualquier organización, especialmente aquellas que dependen de inventarios de productos. La correcta administración de datos constituye un componente fundamental para procesos logísticos, comerciales, de análisis y toma de decisiones.

Dentro de este contexto, en el presente trabajo se abordó el desarrollo de un sistema de gestión de inventario implementado en Python, que permitió demostrar, evaluar y comparar distintos enfoques algorítmicos aplicables al manejo de inventarios digitales. Se enfocó principalmente en tres áreas del procesamiento de datos: carga de información estructurada desde un archivo CSV, ordenamiento de registros según criterios específicos y búsqueda eficiente de productos mediante distintos métodos algorítmicos como búsqueda lineal y binaria, facilitando la comparación de su rendimiento.

Por todo lo mencionado el objetivo de este trabajo fue desarrollar un sistema de gestión de inventario en Python que permita cargar datos desde un archivo CSV, ordenar los productos y realizar búsquedas de forma eficiente. A través de la implementación de algoritmos como Merge Sort, búsqueda binaria y búsqueda lineal, se busca comparar el rendimiento entre distintos métodos de búsqueda midiendo sus tiempos de ejecución.

MARCO TEÓRICO

ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO

Dos operaciones muy importantes en programación de computadoras son: **ordenación y búsqueda**; son esenciales para un gran número de programas de proceso de datos y se estima que en estas operaciones las computadoras insumen gran parte de su tiempo.

1- ORDENAMIENTO

Se considera ordenar como el proceso de reorganizar un conjunto dado de objetos en una secuencia especificada. El objetivo de este proceso es facilitar la búsqueda subsiguiente de los elementos del conjunto ordenado.

TIPOS DE ORDENAMIENTO

1.Quicksort (Ordenamiento Rápido)

Quick Sort también se basa en el paradigma divide y vencerás. Elige un elemento como "pivote" y particiona el arreglo de forma que los elementos menores al pivote queden a la izquierda y los mayores a la derecha, y luego aplica recursivamente el mismo proceso en cada subarreglo.

2. Heap Sort

Construye un heap (montículo) a partir de los datos, y luego extrae el máximo (o mínimo) repetidamente para ordenar el arreglo. En la práctica es más lento que Quick Sort y Merge Sort debido al acceso no secuencial a memoria.

3. Bubble Sort (Ordenamiento Burbuja)

Comparaciones sucesivas de pares de elementos adyacentes, intercambiándolos si están en el orden incorrecto. Este proceso se repite hasta que no se requieran más intercambios.

4. Insertion Sort

Requiere construir gradualmente la lista ordenada tomando un elemento de la lista no ordenada y colocándolo en la posición correcta.

5. Selection Sort

Busca el elemento más pequeño (o grande) y lo coloca al inicio. Repite el proceso con el resto de la lista.

6. Merge Sort:

El algoritmo Merge Sort es uno de los más clásicos y eficientes métodos de ordenamiento, basado en el paradigma "divide y vencerás". Su funcionamiento consiste en dividir recursivamente la lista original en mitades hasta llegar a sublistas de un solo elemento, para luego combinarlas en orden creciente.

Ventajas principales

- **Eficiencia garantizada**: su complejidad temporal es O(n log n) en todos los casos (mejor, promedio y peor).
- **Estabilidad**: mantiene el orden de elementos equivalentes.
- **Aplicaciones reales**: se utiliza en bibliotecas estándar como Python

2- BÚSQUEDA

Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

Existen diferentes métodos de búsqueda:

- Búsqueda Lineal o Secuencial
- Búsqueda Binaria o Dicotómica

Búsqueda Lineal

Esta es la técnica más simple, consiste en recorrer la lista buscando el elemento deseado, desde el primer elemento hasta el último de uno en uno. Si la lista contiene el elemento se devolverá su posición y, en caso contrario, un mensaje que indique el fracaso de la búsqueda. Es notorio el gran inconveniente de este algoritmo, pues sea cual fuere el resultado de la búsqueda (existe / no existe), se recorre la lista completa.

En el caso más desfavorable (el elemento buscado está ubicado al final de la lista o no existe), este método requiere consultar n elementos, por lo tanto, el tiempo de búsqueda es directamente proporcional al número de elementos de la lista.

Características:

- **Complejidad temporal**: O(n), con un mejor caso de O(1) (elemento al inicio) y peor caso de O(n) (elemento al final o inexistente).
- **No requiere ordenamiento previo**.
- **Facilidad de implementación**: no requiere estructuras auxiliares ni conocimientos avanzados.

Aplicaciones:

- Listas pequeñas o con datos que cambian frecuentemente.

Búsqueda Binaria

Este algoritmo funciona efectivamente en vectores que, a diferencia del lineal, están ordenados. Este algoritmo funciona eliminando la mitad de los elementos en cada iteración: comparará el valor buscado con el primer tramo del vector ordenado, y si está dentro se elimina la otra mitad del array. Si no se encuentra en esta mitad, realizará lo propio con la otra mitad. Este proceso lo iterará tantas veces como necesite hasta encontrar el valor. Sin embargo, necesitará menos iteraciones que el algoritmo secuencial, aunque necesitará que el vector este ordenador de alguna forma.

Características clave

- **Complejidad temporal**: O(log n), significativamente mejor que O(n) en grandes listas.
- **Requiere ordenamiento previo**, lo cual implica un costo inicial si los datos no están ordenados.
- **Implementación**: puede ser iterativa (uso constante de memoria) o recursiva (uso adicional de memoria proporcional a log n).

Ventajas:

- Ideal para conjuntos de datos grandes.
- Utilizado en estructuras de datos como árboles binarios de búsqueda.

En resumen, la búsqueda lineal, por su simplicidad, es buena para listas pequeñas, pero para grandes listas es ineficiente; la búsqueda binaria es el método idóneo. Se base en el conocido concepto divide y vencerás.

3- Comparación Práctica y Elección de Algoritmos

En aplicaciones reales, la elección del algoritmo depende del tamaño de los datos, si están ordenados, y la cantidad de operaciones esperadas.

Resumen Comparativo

Algoritmo	Mejor Caso	Peor Caso	Requiere ordenamiento	Eficiencia
Merge Sort	O(n log n)	O(n log n)	No	Alta
Búsqueda Binaria	O(1)	O(log n)	Sí	Muy alta
Búsqueda Lineal	O(1)	O(n)	No	Baja

- Para grandes volúmenes de datos con muchas búsquedas: ordenar con Merge Sort y usar búsqueda binaria.
- Para listas pequeñas o no ordenadas: se puede aplicar búsqueda lineal

CASO PRÁCTICO

Como se mencionó previamente, el objetivo de este trabajo fue de desarrollar un sistema de gestión de inventario en Python que permita cargar datos desde un archivo CSV, ordenar los productos y realizar búsquedas de forma eficiente. A través de la implementación de algoritmos como Merge Sort, búsqueda binaria y búsqueda lineal, se busca comparar el rendimiento entre distintos métodos de búsqueda midiendo sus tiempos de ejecución.

A continuación, se describe el desarrollo del caso práctico “Gestión de Inventario”:

Paso 1: IMPORTACIÓN DE MÓDULOS

```
1   import csv
2   import time
3   import matplotlib.pyplot as plt
```

➔ Se importan tres módulos fundamentales para el funcionamiento del programa:

- **csv**: Para leer archivos *.csv.*
- **time**: Para medir el tiempo de ejecución de las búsquedas.
- **matplotlib.pyplot**: Para generar gráficos que comparen los tiempos de ejecución.

Paso 2: CARGA DE PRODUCTOS DESDE UN ARCHIVO CSV

```
def cargar_productos_csv(nombre_archivo):
    inventario = []
    with open(nombre_archivo, mode='r', encoding='utf-8') as archivo:
        lector = csv.DictReader(archivo)
        for fila in lector:
            fila["precio"] = float(fila["precio"])  # Convertir precio a número
            inventario.append(fila)
    return inventario
```

➔ Se procede a la carga de productos desde un archivo CSV.

- Se abre el archivo y se utilizan las funcionalidades del módulo csv para leer su contenido.

- ❖ A través de csv.DictReader(), cada fila del archivo se interpreta como un diccionario, utilizando los nombres de las columnas como claves.
- ❖ El campo "precio", se convierte a tipo float para permitir su ordenamiento y comparación numérica.
- ❖ Cada producto se almacena en una lista llamada inventario.

Paso 3: ALGORITMOS DE ORDENAMIENTO

3a- Ordenamiento por nombre *(usando sorted() con lambda)*

```python
def ordenar_por_nombre(lista):
    return sorted(lista, key=lambda x: x["nombre"])
```

➔ Se ordena lista de productos por nombre, usando *sorted* con una función *lambda*.

➔ Permite al usuario visualizar los productos de forma ordenada alfabéticamente.

3b- Ordenamiento por precio *(usando Merge Sort)*

```python
def merge_sort_precio(lista):
    if len(lista) <= 1:
        return lista

    medio = len(lista) // 2
    izquierda = merge_sort_precio(lista[:medio])
    derecha = merge_sort_precio(lista[medio:])

    return merge(izquierda, derecha)

def merge(izquierda, derecha):
    resultado = []
    i = j = 0

    while i < len(izquierda) and j < len(derecha):
        if izquierda[i]["precio"] <= derecha[j]["precio"]:
            resultado.append(izquierda[i])
            i += 1
        else:
            resultado.append(derecha[j])
            j += 1

    resultado.extend(izquierda[i:])
    resultado.extend(derecha[j:])
    return resultado
```

➔ Se ordena la lista de productos por precio, usando *merge sort*, el cual es eficiente e ideal para listas largas.

- ❖ Divide la lista en mitades.
- ❖ Ordena cada mitad de forma recursiva.
- ❖ Une ambas partes ordenadas en una sola lista.

Paso 4: ALGORITMOS DE BÚSQUEDA

4a- Búsqueda binaria por ID

```python
def busqueda_binaria_id(lista_ordenada, id_buscado):
    bajo = 0
    alto = len(lista_ordenada) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista_ordenada[medio]["id"] == id_buscado:
            return lista_ordenada[medio]
        elif lista_ordenada[medio]["id"] < id_buscado:
            bajo = medio + 1
        else:
            alto = medio - 1
    return None
```

Aclaración: La lista de productos se carga desde un archivo CSV que ya se encuentra previamente ordenado por ID. No obstante, durante el programa se vuelve a ordenar por precaución antes de aplicar este algoritmo.

➔ La búsqueda binaria localiza el producto por su ID de forma eficiente, dividiendo la lista ordenada por la mitad en cada paso, hasta encontrar el producto buscado (o determinar su ausencia), lo que permite una búsqueda rápida.

- ❖ **Ventaja**: muy eficiente en listas ordenadas. Su complejidad es O(log n), lo cual permite encontrar elementos rápidamente incluso en listas muy grandes.

4b- Búsqueda lineal por ID

```python
def busqueda_lineal_id(lista, id_buscado):
    for producto in lista:
        if producto["id"] == id_buscado:
            return producto
    return None
```

➔ La búsqueda lineal recorre la lista desde el inicio (comparando uno por uno los elementos) hasta encontrar el producto con el ID buscado (o llegar al final sin éxito).

- ❖ Ventaja: no requiere que la lista esté ordenada.
- ❖ Desventaja: más lenta con listas grandes, ya que su complejidad es O(n), lo que implica que el tiempo de búsqueda crece proporcionalmente con la cantidad de elementos.

PROGRAMA PRINCIPAL

Paso 5: CARGA INICIAL DEL PROGRAMA

```
print("Bienvenido al sistema de inventario.\n")

# Cargar inventario desde CSV
nombre_csv = input("Ingrese el nombre del archivo CSV (ej: productos.csv): ")
inventario = cargar_productos_csv(nombre_csv)
```

➔ Se imprime un mensaje de bienvenida.

➔ Luego se solicita al usuario ingresar el nombre del archivo CSV y se carga el inventario desde dicho archivo.

Paso 6: VERIFICACIÓN DE ORDEN POR ID *(para búsqueda binaria)*

```
inventario_ordenado_por_id = sorted(inventario, key=lambda x: x["id"])
```

➔ Se ordena una vez por ID para permitir la búsqueda binaria.

- ❖ Aunque el archivo CSV se encuentra previamente ordenado por ID, se vuelve a ordenar por precaución para asegurar el correcto funcionamiento de la búsqueda binaria.
- ❖ Esta medida asegura que la lista esté efectivamente ordenada y resulta útil en caso de que el archivo CSV sea modificado (por ejemplo, si se agregan, eliminan o reordenan productos), ya que esos cambios podrían alterar el orden original.

Paso 7: APLICACIÓN DE ORDENAMIENTO *(en el programa)*

7a- Selección y ejecución del criterio de Ordenamiento

```
opcion = input("¿Desea ordenar los productos por (1) nombre o (2) precio? Ingrese 1 o 2: ")
if opcion == "1":
    ordenados = ordenar_por_nombre(inventario)
    print("\nProductos ordenados por nombre:")
    for prod in ordenados[:50]:  # Muestra solo los primeros 50
        print(prod)
elif opcion == "2":
    inventario = merge_sort_precio(inventario)
    print("\nProductos ordenados por precio:")
    for prod in inventario[:50]:  # Muestra solo los primeros 50
        print(prod)
else:
    print("Opción inválida.")
```

Según la elección del usuario:

- ❖ Si elige 1, se ordenan los productos por nombre (alfabéticamente).
- ❖ Si elige 2, se ordenan los productos por precio.

Se muestran los primeros 50 productos como verificación (en cada caso).

Paso 8: APLICACIÓN Y ANÁLISIS DE BÚSQUEDA

8a- Ingreso de ID y medición de tiempos de búsqueda

```
id_buscado = input("\nIngrese el ID del producto que desea buscar (ej: P002): ")

repeticiones = 100
tiempos_binaria = []
tiempos_lineal = []
```

```
for _ in range(repeticiones):
    inicio_bin = time.perf_counter()
    busqueda_binaria_id(inventario_ordenado_por_id, id_buscado)
    fin_bin = time.perf_counter()
    tiempos_binaria.append(fin_bin - inicio_bin)

    inicio_lin = time.perf_counter()
    busqueda_lineal_id(inventario, id_buscado)
    fin_lin = time.perf_counter()
    tiempos_lineal.append(fin_lin - inicio_lin)
```

- ➔ Se solicita al usuario qué producto desea buscar por ID.
- ➔ Se repiten ambas búsquedas (binaria y lineal) 100 veces para obtener una estimación representativa.
- ➔ Se mide el tiempo de cada ejecución con time.perf_counter() y se almacenan los resultados en listas separadas.

8b- Promedio tiempos y resultados de la búsqueda

```
promedio_binaria = sum(tiempos_binaria) / repeticiones
promedio_lineal = sum(tiempos_lineal) / repeticiones
```

Se calcula el tiempo promedio de cada algoritmo de búsqueda en base a las repeticiones.

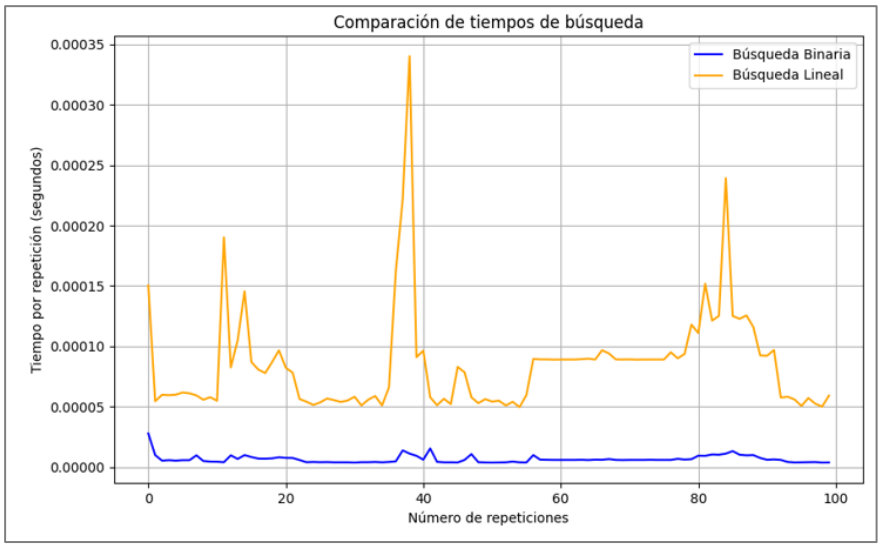
```
resultado_bin = busqueda_binaria_id(inventario_ordenado_por_id, id_buscado)
resultado_lin = busqueda_lineal_id(inventario, id_buscado)

print("\nResultado de búsqueda binaria:")
if resultado_bin:
    print(resultado_bin)
else:
    print("No encontrado.")

print(f"Tiempo promedio búsqueda binaria: {promedio_binaria:.10f} segundos")

print("\nResultado de búsqueda lineal:")
if resultado_lin:
    print(resultado_lin)
else:
    print("No encontrado.")

print(f"Tiempo promedio búsqueda lineal: {promedio_lineal:.10f} segundos")
```

- ➔ Se muestra si el producto fue encontrado o no.
- ➔ Se imprimen los tiempos promedios de cada tipo de búsqueda, lo que permite comparar cuál es más rápida.
- ➔ Luego se imprimen los resultados de ambas búsquedas.

8c- Visualización gráfica de la comparación de los tiempos

```
plt.figure(figsize=(10,6))
plt.plot(tiempos_binaria, label="Búsqueda Binaria", color='blue')
plt.plot(tiempos_lineal, label="Búsqueda Lineal", color='orange')
plt.xlabel('Número de repeticiones')
plt.ylabel('Tiempo por repetición (segundos)')
plt.title('Comparación de tiempos de búsqueda')
plt.legend()
plt.grid(True)
plt.show()
```

- ➔ Se utiliza Matplotlib para graficar la evolución del tiempo de ejecución en función de las repeticiones, permitiendo visualizar de manera clara y comparativa la eficiencia de ambos algoritmos de búsqueda.



➔ Visualización gráfica de los resultados

➔ Se genera un gráfico de líneas con:

Eje X: número de repeticiones

Eje Y: tiempo de ejecución de cada búsqueda

METODOLOGÍA UTILIZADA

El desarrollo del programa se llevó a cabo siguiendo una metodología estructurada, mediante la implementación y el análisis de algoritmos de búsqueda y ordenamiento aplicados a un sistema de inventario, utilizando el lenguaje Python.

En primer lugar, se definió el objetivo: gestionar una lista de productos de una tienda online. Se abordaron cada uno de los aspectos planteados: *carga de datos, ordenamiento, búsqueda, análisis de eficiencia y visualización comparativa*.

A continuación, se describen los pasos aplicados:

1. Diseño del modelo de datos

Se recopiló un conjunto de datos representativos en formato CSV, con 500 productos ficticios estructurados con los siguientes campos:

- **id**: cadena alfanumérica única para identificar el producto.
- **nombre**: texto en minúsculas que representa el nombre del producto.
- **categoría** (8 categorías predefinidas)
- **precio**: valor numérico correspondiente al precio del producto.

2. Carga de datos desde archivo CSV

Se utilizó la biblioteca *csv* de Python para leer y estructurar los datos en una lista de diccionarios.

Función: *def cargar_productos_csv(nombre_archivo)*

Esta función abrió y leyó el archivo *.csv* utilizando *csv.DictReader*. Cada fila fue convertida en un diccionario y el campo precio fue transformado a tipo float para permitir operaciones numéricas y comparaciones.

Como resultado se obtuvo una lista de 500 productos representados como diccionarios.

3. Implementación de funciones (ordenamiento y búsqueda)

a- Ordenamiento de productos

Se incluyeron dos métodos para ordenar los productos, uno por nombre y otro por precio:

- **Ordenamiento por nombre**

  Se utilizó la función sorted() con una función lambda. Lo que permitió ordenar la lista de productos alfabéticamente por el campo "nombre".

  Este enfoque utiliza el algoritmo Timsort, que tiene una complejidad de O(n log n), lo que contribuye a un buen rendimiento incluso en listas parcialmente ordenadas.

- **Ordenamiento por precio**

  Se implementó el algoritmo Merge Sort. Esta función permitió ordenar la lista de productos por el campo "precio". Este algoritmo divide la lista en mitades, ordena recursivamente y luego fusiona las sublistas ya ordenadas en una sola lista. Este enfoque presenta un rendimiento consistente con complejidad O(n log n), lo que resulta ideal para listas de gran tamaño.

De esta forma el programa brindó al usuario la opción de ordenar los productos según su elección (por nombre o precio).

b- Búsqueda de productos por ID

Se desarrollaron dos métodos para buscar productos por su ID

- **Búsqueda lineal** *(def busqueda_lineal_id(lista, id_buscado))*

  Recorre la lista secuencialmente hasta encontrar el producto con el ID buscado. Su complejidad es O(n), por lo que puede volverse ineficiente en listas grandes.

- **Búsqueda binaria** *(def busqueda_binaria_id(lista_ordenada, id_buscado))*

  Busca el producto usando el algoritmo de búsqueda binaria, el cual requirió una lista ordenada previamente por ID. Es mucho más rápida que la búsqueda lineal (tiene complejidad de O(log n)) siendo significativamente más eficiente en listas grandes.

4. Medición del tiempo de ejecución

Se utilizó *time.perf_counter()* para realizar mediciones precisas del tiempo de ejecución de cada tipo de búsqueda (binaria y lineal).

Las pruebas se repitieron 100 veces para obtener promedios representativos del comportamiento de cada algoritmo.
Se registraron los tiempos de ejecución individuales para cada repetición en ambas búsquedas.

5. Visualización de resultados

Para comparar visualmente la eficiencia de ambos algoritmos de búsqueda, se utilizó el módulo matplotlib.pyplot. Se graficaron los tiempos de ejecución por repetición para ambos métodos (búsqueda lineal y binaria), lo cual permitió realizar un análisis comparativo.

Por último, se mencionan las herramientas utilizadas durante el trabajo:

Herramientas y recursos utilizados

- Python 3.11.9
- Visual Studio Code
- Librería Matplotlib
- Libreria Time
- Librería CSV

RESULTADOS

Durante la prueba del programa, se obtuvieron los siguientes resultados:

1. Carga y estructura de datos

Al iniciar el programa, los 500 productos fueron cargados correctamente desde el archivo CSV, conformando una lista de diccionarios. Se verificó que todos los campos estuvieran en el tipo de dato adecuado, lo cual permitió continuar sin errores en las etapas posteriores.

2. Ordenamiento de productos

Se probaron ambos métodos de ordenamiento implementados:

- **Por nombre** *(sorted + lambda):* el listado fue ordenado alfabéticamente de forma inmediata y sin errores, como prueba se mostraron los primero 50 productos.
- **Por precio** *(Merge Sort):* el algoritmo realizó el ordenamiento correctamente. Se observó un comportamiento estable, validando la idoneidad del algoritmo para este tipo de datos.

3. Búsqueda por ID: Comparación de algoritmos

Se realizaron 100 ejecuciones repetidas de cada tipo de búsqueda para medir tiempos de respuesta.

- **Búsqueda Binaria:** demostró ser notablemente más rápida, con tiempos de ejecución bajos y consistentes en cada repetición. Esto se reflejó en una línea prácticamente horizontal en la visualización gráfica (figura 1 y 2).
- **Búsqueda Lineal:** presentó mayor variabilidad y tiempos significativamente más altos, observándose ciertos picos en los gráficos.

4. Visualización gráfica y análisis

Los resultados obtenidos se representaron mediante un gráfico comparativo generado con Matplotlib, en el que se visualizó el tiempo que tarda cada algoritmo en cada repetición. La línea azul (búsqueda binaria) mostró un comportamiento estable, mientras que la línea naranja (búsqueda lineal) evidenció picos más altos.

- La **línea azul**, correspondiente a la *búsqueda binaria*, se observaron valores de tiempo bajos y constantes.
- La **línea naranja**, correspondiente a la *búsqueda lineal*, se evidenciaron valores significativamente más altos y con mayor variabilidad.

Figura 1. Comparación de tiempos de búsqueda para el producto con ID "P250"

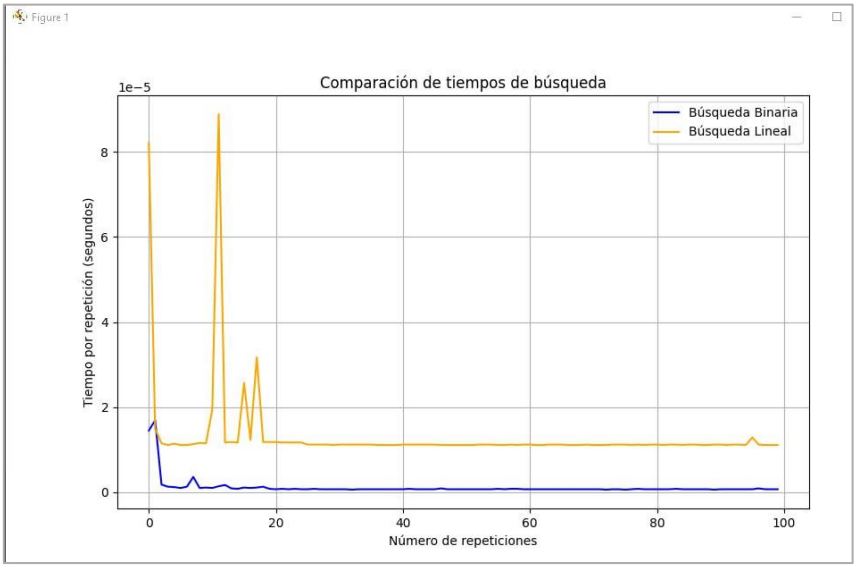
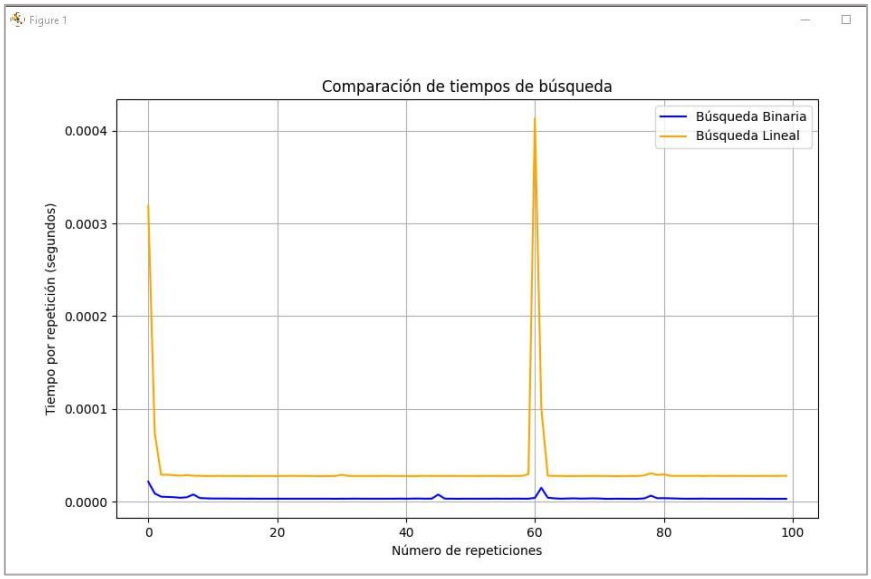


Figura 2. Comparación de tiempos de búsqueda para el producto con ID "P480"



Este comportamiento confirma la teoría de que la búsqueda binaria resulta ser superior en términos de eficiencia, siempre y cuando se cumpla la condición de que la lista esté previamente ordenada.

CONCLUSIÓN

La realización de este trabajo integrador permitió reforzar conocimientos fundamentales sobre algoritmos de búsqueda y ordenamiento, así como su aplicación en un caso real, como la gestión de un inventario de una tienda online.

Se pudo constatar la diferencia en eficiencia entre búsqueda lineal y búsqueda binaria, especialmente al trabajar con grandes volúmenes de datos. La búsqueda binaria demostró ser notablemente más rápida, aunque requiere una condición: tener la lista previamente ordenada. Esto permitió comprender la importancia de elegir el algoritmo adecuado en función de los requisitos y características de los datos.

Además, se adquirió experiencia en el manejo de archivos CSV, el uso de bibliotecas de Python como csv, matplotlib y time, herramientas utilizadas en el desarrollo de software y análisis de datos.

Durante la práctica, surgieron dificultades menores al inicio, vinculadas a la manipulación del archivo CSV y el uso de bibliotecas como matplotlib y time, debido a cuestiones de sintaxis o configuración. Estas dificultades fueron superadas mediante investigación, pruebas repetidas y consulta de la documentación.

Como posibles mejoras, se podría incorporar funciones de filtrado por categoría o rango de precios, exportar los resultados (tanto textos como gráficas) a formatos como csv o imágenes, para su inclusión en reportes o presentaciones y extender el análisis comparativo a otros algoritmos de ordenamiento o estructuras de datos.

BIBLIOGRAFÍA

- Ankit, A., et al. (2017). A study on the optimization of binary search algorithm. IIARD International Journal of Computer Applications.

- GeeksforGeeks. (n.d.). Searching algorithms.

- González de la Lama, S. (2024). Aplicación de algoritmos de búsqueda en la optimización de caminos mínimos en grafos de decisión (Trabajo Fin de Grado, Universidad Politécnica de Madrid). Escuela Técnica Superior de Ingenieros Informáticos.

- Knuth, D. E. (1998). The art of computer programming: Sorting and searching (Vol. 3). Addison-Wesley.

- MIT OpenCourseWare. (n.d.). Introduction to algorithms. https://ocw.mit.edu

- Parmar, D., & Kumbharana, C. (2015). A comparative study of linear and binary search algorithms. International Journal of Computer Applications, 113(1).

- Universidad Nacional de Salta, Facultad de Ciencias Exactas. (2010). Elementos de programación: Apuntes de cátedra (Licenciatura en Análisis de Sistemas, Plan 2010). Universidad Nacional de Salta.

- Zinnia, N. A. (2024). An efficient approach of binary search algorithm. arXiv:2403.15825. https://arxiv.org/abs/2403.15825