Algoritmos de Búsqueda y Ordenamiento

Filtros de Búsqueda y ordenamiento en Tiendas Online

Alumnos

Matias Luna - matiluna09@gmail.com
Lautaro Marin - marin.lautaro.ibqm4b@gmail.com

Materia

Programación 1

Comisión

16

Profesor

Cinthia Rigoni

Fecha de Entrega

8 de junio 2025

Índice

1- Introducción

El tema seleccionado para esta investigación es sobre los Filtros de búsqueda y ordenamiento en Tiendas Online, principalmente porque es un recursos ampliamente distribuido en las plataformas de venta en línea, se estima que en 2024 las ventas en línea representaron un total de 2.71 millones aproximadamente, contribuyendo estos filtros de búsqueda y ordenamiento a optimizar el proceso de compras, lograr una óptima experiencia de usuario.

Estas herramientas permiten a los usuarios encontrar productos de manera rápida y precisa dentro de grandes catálogos, facilitando el proceso de decisión de compra. Por esto es fundamental entender los algoritmos de búsqueda y ordenamiento que hacen posible estas funcionalidades, ya que su correcta implementación mejora el rendimiento y la usabilidad de una tienda online.

El objetivo de este trabajo es investigar y comprender los algoritmos que sustentan los filtros de búsqueda y ordenamiento, analizando en profundidad su funcionamiento, ventajas, limitaciones y complejidad temporal. Asimismo, se busca evaluar su aplicación práctica en el contexto de una tienda online, simulando situaciones reales de interacción con el usuario.

2 - Marco Teórico

Los filtros y formas de ordenamiento que la gran mayoría de tiendas en línea presentan son:

- Búsqueda por nombre (Búsqueda lineal o binaria)
- Filtrar por categoría (Búsqueda)
- Ordenar por precio (Ordenamiento)

Búsqueda por nombre

Una de las funcionalidades más utilizadas en las tiendas en línea es permitir al usuario buscar un producto por su nombre.

Búsqueda lineal

Este tipo de búsqueda se aplica principalmente cuando:

- La lista de productos es pequeña.
- Los productos no están ordenados alfabéticamente.

La búsqueda lineal consiste en recorrer uno por uno todos los elementos de la lista y comparar el nombre del producto con el texto ingresado por el usuario. Es fácil de implementar, pero poco eficiente para grandes volúmenes de datos, ya que en el peor caso revisa todos los elementos.

Complejidad

La complejidad temporal de la búsqueda lineal es de *O(n)* siendo *n* el número de elementos (Productos) en la lista. En el peor de los casos el algoritmo tendrá que comprobar cada elemento de la lista antes de encontrar el elemento deseado.

Búsqueda binaria

Se utiliza cuando:

- Los productos están ordenados alfabéticamente por su nombre.

En este caso, se aplica la búsqueda binaria, que reduce el espacio de búsqueda a la mitad en cada paso, comparando el valor medio con el nombre buscado. Es mucho más rápida, especialmente cuando se manejan cientos o miles de productos, pero requiere que la lista esté previamente ordenada.

Aplicación en tiendas online

Muchas tiendas tienen un campo de búsqueda que primero ordena alfabéticamente los productos y luego realiza una búsqueda binaria o accede directamente con índices u otras estructuras eficientes. El objetivo es que el usuario obtenga resultados al instante al escribir el nombre de un producto.

Complejidad

La complejidad temporal de la búsqueda binaria es de *O(log n)*, haciendo que este algoritmo sea más efectivo que el algoritmo de búsqueda lineal ya que reduce el espacio de búsqueda a la mitad en cada paso.

Filtrar por categoría

Otra función esencial es permitir que el usuario filtre los productos según la categoría.

Búsqueda lineal

El filtrado por categoría consiste en:

- Recorrer todos los productos de la lista.
- Verificar si la categoría del producto coincide con la categoría seleccionada.

Este tipo de búsqueda se basa en una condición lógica que compara valores. Aunque no es la más eficiente, es sencilla y útil cuando la cantidad de productos no es demasiado alta.

Aplicación en tiendas online

La mayoría de las plataformas ya organizan sus datos para facilitar este tipo de filtrado, permitiendo que el usuario seleccione una categoría y vea los resultados de forma inmediata. Sin embargo la búsqueda lineal es una forma válida de implementar esta funcionalidad.

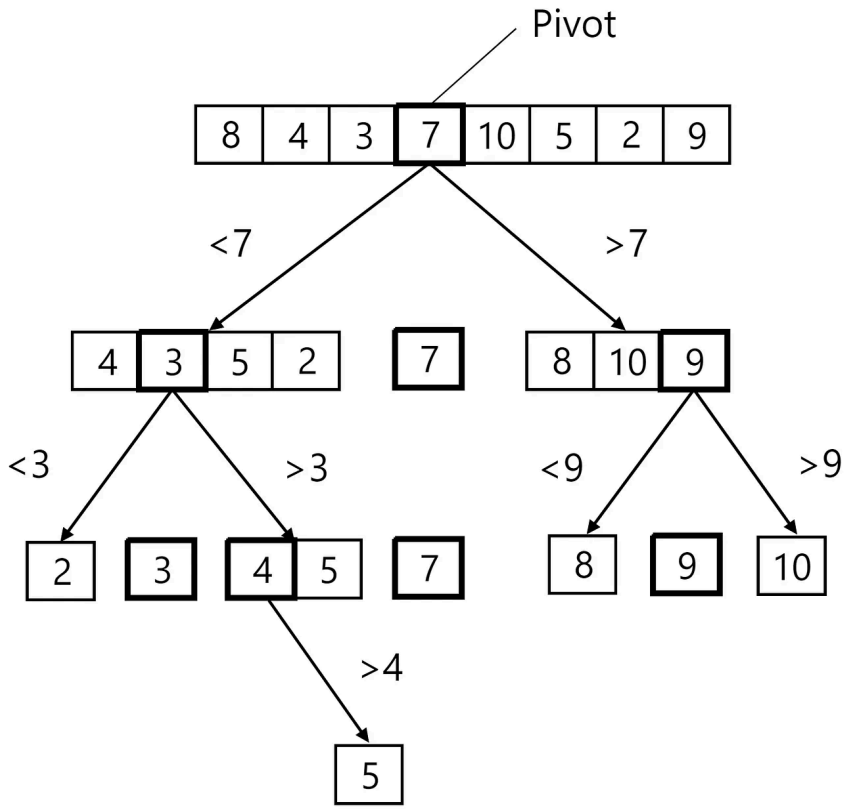
Complejidad

Como dijimos anteriormente en la explicación anterior de este algoritmo, la complejidad temporal de la búsqueda lineal es de *O(n)*.

Ordenar por precio

Para ordenar los productos por precio en forma ascendente se utiliza un algoritmo de ordenamiento, en general se utiliza Quick Sort y Merge Sort ya que son algoritmos eficientes.

Algoritmo Quick Sort



Quick Sort es un algoritmo de búsqueda de ordenamiento basado en el principio “Divide y Vencerás”, dividiendo el problema en subproblemas más pequeños y fáciles de manejar lo que lo hace un algoritmo eficiente en grandes conjuntos de datos y requiere una pequeña cantidad de memoria para funcionar.

Complejidad

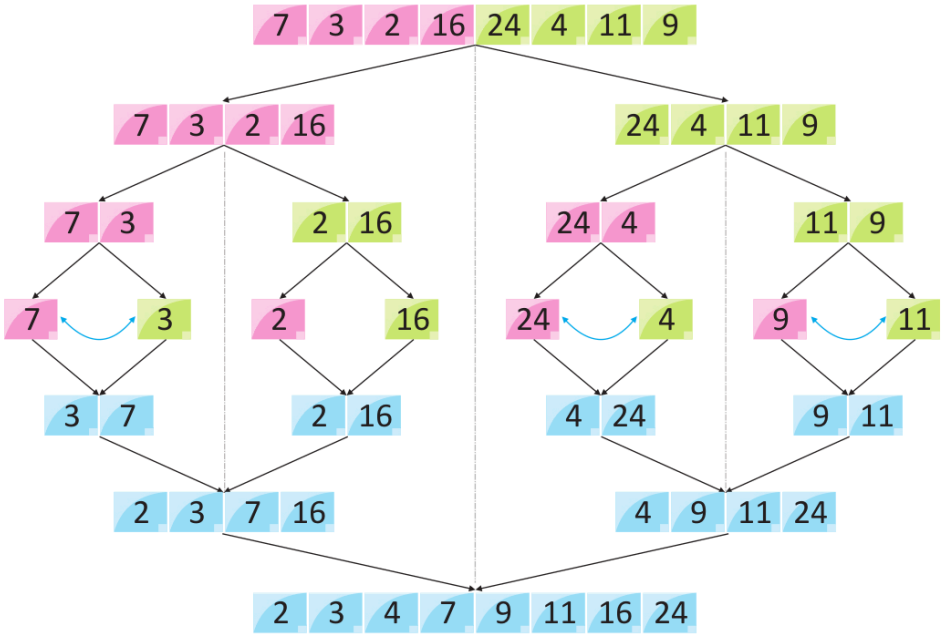
- El mejor caso tiene una complejidad de *(Ω(n log n))*, que ocurre cuando el elemento pivote divide la matriz en dos mitades iguales
- El caso promedio tiene una complejidad de *((θ(n log n))*, que ocurre cuando el pivote divide la matriz en dos mitades pero no tienen la misma longitud

- El peor caso tiene la complejidad de *(O(n²))* sucede cuando el elemento más grande o más pequeño se utiliza como pivote.

Aplicación en Filtros de ordenamiento por precios

Para aplicar Quick sort en un filtro de ordenamiento por precio se elige un producto como pivote y se tiene en cuenta el precio del mismo, luego se divide la lista de productos en dos grupos, los que tienen un precio menor que el producto pivote y los que tienen un precio mayor, para posteriormente ordenar de forma recursiva cada grupo

Algoritmo Merge Sort



Merge Sort es un algoritmo de ordenamiento, que al igual que Quick Sort, se basa en el principio de “Divide y vencerás”, diferenciándose de este porque no se selecciona un elemento como pivote, sino que se divide la lista en dos mitades, ordenandolas recursivamente y para luego fusionarlas en una única lista ordenada.

Complejidad

En Merge Sort la complejidad temporal de *O(n log n)* en todos los casos (Mejor, promedio y peor) ya que el proceso de división y fusión se realiza de manera constante, sin depender del orden inicial de los elementos.

Aplicación en Filtros de ordenamiento por precios

Para aplicar Merge Sort en un filtro de ordenamiento por precio se divide la lista de productos en dos mitades de forma recursiva, hasta que cada sublista tenga un solo producto, luego se ordena de forma recursiva hasta llegar a elementos individuales para posteriormente comparar los precios de los productos en sublistas y combinando en una nueva lista de forma ordenada con precios ascendentes.

3 - Caso Práctico:

Productos

Se declaró una lista de 10 productos base para hacer las pruebas

```
productos = [
    {
        "nombre": "Smartphone Negro",
        "categoria": "Electrónica",
        "precio": 999.99
    },
    {
        "nombre": "Zapatillas deportivas",
        "categoria": "Calzado deportivo",
        "precio": 129.99
    },
    {
        "nombre": "Cafetera Express",
        "categoria": "Electrodomésticos",
        "precio": 299.50
    },
    {
        "nombre": "Libro de Programación",
        "categoria": "Libros",
        "precio": 45.99
    },
    {
        "nombre": "Monitor para Computadora",
        "categoria": "Electrónica",
        "precio": 349.99
    },
    {
        "nombre": "Silla Gamer",
        "categoria": "Muebles",
        "precio": 199.99
    },
    {
        "nombre": "Cafetera Express Chica",
        "categoria": "Electrodomésticos",
        "precio": 89.99
    },
    {
        "nombre": "Mochila",
        "categoria": "Accesorios",
        "precio": 59.99
    },
    {
        "nombre": "Reloj Smartwatch",
        "categoria": "Electrónica",
        "precio": 199.50
    },
    {
        "nombre": "Auriculares Bluetooth",
        "categoria": "Electrónica",
        "precio": 149.99
    }
]
```

Búsqueda por nombre

integrador_programacion > buscar_nombre.py > buscar_por_nombre

```python
from productos import productos

def busqueda_lineal(nombre_buscado: str):
    for producto in productos:
        if producto["nombre"].lower() == nombre_buscado.lower():
            return producto
    return None

def busqueda_binaria(nombre_buscado: str):
    productos_ordenados = sorted(productos, key=lambda x: x["nombre"].lower())
    izquierda = 0
    derecha = len(productos_ordenados) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        actual = productos_ordenados[medio]["nombre"].lower()

        if actual == nombre_buscado.lower():
            return productos_ordenados[medio]
        elif nombre_buscado.lower() < actual:
            derecha = medio - 1
        else:
            izquierda = medio + 1

    return None

def buscar_por_nombre():
    nombre = input("Ingresa el nombre exacto del producto: ")
    metodo = int(input("Seleccioná el método de búsqueda:\n1) Búsqueda Lineal\n2) Búsqueda Binaria (requiere lista ordenada)\n"))

    if metodo == 1:
        resultado = busqueda_lineal(nombre)
    elif metodo == 2:
        resultado = busqueda_binaria(nombre)
    else:
        print("No ingresaste el nombre correcto")
        return buscar_por_nombre()

    if resultado:
        print("Tu producto:")
        print(resultado)
    else:
        print("Producto no encontrado.")
```

Se importa la lista de productos desde el archivo productos.py.
Esa lista contiene los productos con sus atributos como nombre, categoría y precio

busqueda_lineal(nombre_buscado: str): busca un producto por su nombre usando el algoritmo de búsqueda lineal.
Funcionamiento: Recorre todos los productos uno por uno.
Compara el nombre del producto con el buscado.
Si encuentra coincidencia, lo retorna. Si no, retorna None.

busqueda_binaria(nombre_buscado: str): busca un producto por nombre usando búsqueda binaria, que es más eficiente pero requiere que la lista esté ordenada alfabéticamente.
Funcionamiento:
Ordena la lista de productos por nombre (sorted(...)).
Inicializa dos variables: izquierda y derecha para definir los extremos de búsqueda.
Calcula el punto medio, compara el nombre, y según si es mayor o menor:

Ajusta el rango de búsqueda (mitad izquierda o derecha).
Si encuentra coincidencia, retorna el producto.
Si no lo encuentra después de terminar el bucle, retorna None.

buscar_por_nombre(): Pide al usuario el nombre del producto a buscar.
El tipo de búsqueda que desea usar:
1 = Lineal o 2 = Binária

Luego el **if** ejecuta la búsqueda usando la función correspondiente según la opción ingresada. Y si el usuario ingresa una opción inválida, vuelve a ejecutar la función. O muestra el producto encontrado o informa si no existe.

Resultado final por consola

```
PS D:\Desktop\TP Programacion 1> & C:/Users/Mati/AppData/Local/Microsoft/WindowsApps/python3.11.exe "d:/Desktop/TP Programacion 1/integrador_programacion/buscar_nombre.py"
Ingresa el nombre exacto del producto: mochila
Seleccioná el método de búsqueda:
1) Búsqueda Lineal
2) Búsqueda Binaria (requiere lista ordenada)
1
Tu producto:
{'nombre': 'Mochila', 'categoria': 'Accesorios', 'precio': 59.99}
PS D:\Desktop\TP Programacion 1>
```

Filtrar por categoría

integrador_programacion > filtrar_categoria.py > ejecutar_filtrado_categoria

```python
from productos import productos

def filtrar_por_categoria(categoria_buscada: str):
    return [producto for producto in productos if producto["categoria"].lower() == categoria_buscada.lower()]

def ejecutar_filtrado_categoria():
    categoria = input("Ingresá la categoría: ")
    resultados = filtrar_por_categoria(categoria)

    if resultados:
        print(f"\nProductos en la categoría '{categoria}':\n")
        for producto in resultados:
            print(producto)
    else:
        print(f"No se encontraron productos en la categoría '{categoria}'.")

ejecutar_filtrado_categoria()
```

Se importa la lista de productos desde el archivo productos.py.
Esa lista contiene los productos con sus atributos como nombre, categoría y precio

filtrar_por_categoria(categoria_buscada): devuelve una nueva lista con todos los productos que pertenecen a una categoría determinada.
categoria_buscada es la categoría ingresada por el usuario.

Funcionamiento interno

Utiliza una lista.
Convierte ambas cadenas (la del producto y la buscada) a minúsculas con .lower() para que la comparación no dependa de mayúsculas/minúsculas.
Solo se agregan los productos de la categoría ingresada.

ejecutar_filtrado_categoria(): Pide al usuario que escriba la categoría deseada y llama a la función filtrar_por_categoria y guarda el resultado en resultados.

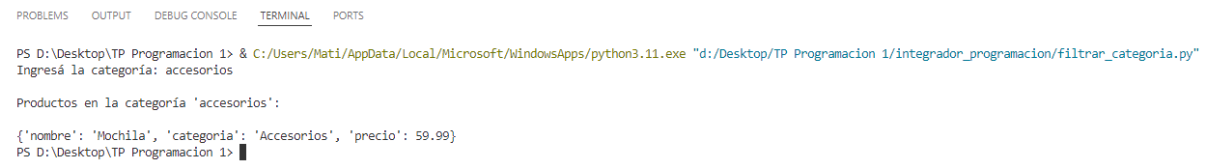
Si hay resultados

Imprime un mensaje con la categoría buscada.
Recorre la lista de productos filtrados y los muestra uno por uno.

Si el usuario no escribe la categoría existente

Informa que no se encontraron productos en esta categoría.

Resultado final por consola



Ordenar por precio / Quick Sort

```python
def quick_sort(productos: list):

    #Si la lista tiene 1 elemento o menos ya se encuentra ordenada
    if len(productos) <= 1:
        return productos

    #Se toma el elemento del medio como pivote
    pivote = productos.pop(len(productos) // 2)

    #Se declara las listas de precios
    precio_alto = []
    precio_bajo = []

    #Se recorre la lista de productos y con un ternario evaluamos el precio
    #precio_alto: Se agrega a la lista si el precio del producto actual es mayor al precio del pivote
    #precio_bajo: Se agrega a la lista si el precio del producto actual es mas bajo que el precio del pivote

    for producto in productos:
        precio_alto.append(producto) if producto["precio"] >= pivote["precio"] else precio_bajo.append(producto)

    #Se concatena todos las listas de precios utilizando recursivamente quick sort junto con el pivote
    return quick_sort(precio_bajo) + [pivote] + quick_sort(precio_alto)
```

Se declara la función quick_sort que recibe como parámetro una lista de productos

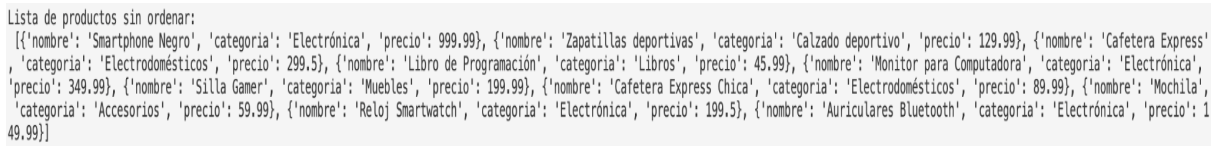
- Si la lista de productos tiene un elemento o menos ya se considera que se encuentra ordenada y se retorna la lista de productos sin modificar

- Caso contrario se toma el producto del medio como pivote y se declaran dos variables para almacenar los productos con precio más bajo o más alto que el precio del producto tomado como pivote. Posteriormente recorre la lista de productos, comparando el precio del producto a evaluar con el precio del pivote, guardando en su respectiva lista dependiendo la condición a evaluar.

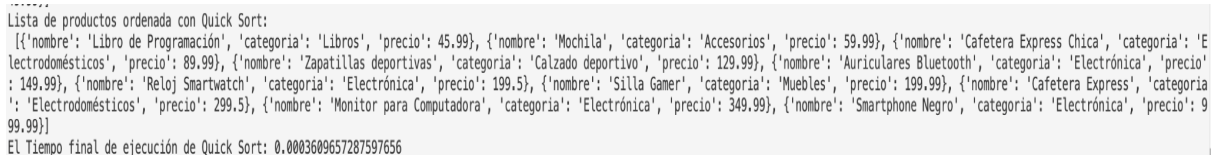
  Para finalizar se retorna el resultado de ordenar recursivamente la lista de precios bajos, concatenado con el pivote y el orden recursivo de los precios más altos

Resultados obtenidos

Lista de productos antes de ordenarla



Lista de productos luego de aplicar Quick Sort



Ordenar por precio / Merge Sort

```python
def merge(primer_lista : list, segunda_lista : list):

    resultado = []
    indice = [0,0] #indice de primera y segunda lista

    while indice[0] < len(primer_lista) and indice[1] < len(segunda_lista):
        if primer_lista[indice[0]]["precio"] < segunda_lista[indice[1]]["precio"]:
            resultado.append(primer_lista[indice[0]])
            indice[0] += 1
        else:
            resultado.append(segunda_lista[indice[1]])
            indice[1] += 1

    # Verificar elementos restantes en la primer lista
    while indice[0] < len(primer_lista):
        resultado.append(primer_lista[indice[0]])
        indice[0] += 1

    # Verificar elementos restantes en la segunda lista
    while indice[1] < len(segunda_lista):
        resultado.append(segunda_lista[indice[1]])
        indice[1] += 1

    return resultado
```

Se declara la función de merge que recibe como parámetro dos listas de productos, se inicializan dos variables, una para guardar el resultado y otra para los índices de las listas, posteriormente se recorre ambas listas hasta que el índice ambas listas sea mayor a la longitud de estas, comparando en cada iteración el precio de cada lista según su indice.

Si el precio del producto de la primera lista es menor que el de la segunda lista se agrega este producto al resultado y se suma uno a el índice de la lista correspondiente, caso contrario, se agrega el producto de la segunda lista.

Luego, se verifica si quedan elementos restantes en alguna de las dos listas y se agrega el resultado de forma secuencial.

```python
def merge_sort(productos: list):

    #Si la lista tiene 1 elemento o menos ya se encuentra ordenada
    if len(productos) <= 1:
        return productos

    #Se divide la lista de productos en dos mitades.
    primer_lista = productos[:len(productos)//2]
    segunda_lista = productos[len(productos)//2:]

    #Ordenamos de forma recursiva cada mitad
    primer_lista = merge_sort(primer_lista)
    segunda_lista = merge_sort(segunda_lista)

    return merge(primer_lista, segunda_lista)
```

Se declara la función merge_sort la cual recibe como parámetro una lista de productos, si la lista de productos tiene un producto o menos ya se encuentra ordenada por lo que retorna la misma lista de producto. Caso contrario se divide la lista de productos en dos mitades, para posteriormente ordenar de forma recursiva cada mitad.

Para finalizar utiliza la función merge previamente declarada que recibe como parámetros las dos partes de la lista divida, generando así la lista de productos ordenada.

Resultados obtenidos

Lista de productos antes de ordenarla

```
Lista de productos sin ordenar:
 [{'nombre': 'Smartphone Negro', 'categoria': 'Electrónica', 'precio': 999.99}, {'nombre': 'Zapatillas deportivas', 'categoria': 'Calzado deportivo', 'precio': 129.99}, {'nombre': 'Cafetera Express'
, 'categoria': 'Electrodomésticos', 'precio': 299.5}, {'nombre': 'Libro de Programación', 'categoria': 'Libros', 'precio': 45.99}, {'nombre': 'Monitor para Computadora', 'categoria': 'Electrónica',
'precio': 349.99}, {'nombre': 'Silla Gamer', 'categoria': 'Muebles', 'precio': 199.99}, {'nombre': 'Cafetera Express Chica', 'categoria': 'Electrodomésticos', 'precio': 89.99}, {'nombre': 'Mochila',
 'categoria': 'Accesorios', 'precio': 59.99}, {'nombre': 'Reloj Smartwatch', 'categoria': 'Electrónica', 'precio': 199.5}, {'nombre': 'Auriculares Bluetooth', 'categoria': 'Electrónica', 'precio': 1
49.99}]
```

Lista de productos después de aplicar Merge Sort

```
Lista de productos ordenada con Merge Sort:
 [{'nombre': 'Libro de Programación', 'categoria': 'Libros', 'precio': 45.99}, {'nombre': 'Mochila', 'categoria': 'Accesorios', 'precio': 59.99}, {'nombre': 'Cafetera Express Chica', 'categoria': 'E
lectrodomésticos', 'precio': 89.99}, {'nombre': 'Zapatillas deportivas', 'categoria': 'Calzado deportivo', 'precio': 129.99}, {'nombre': 'Auriculares Bluetooth', 'categoria': 'Electrónica', 'precio'
: 149.99}, {'nombre': 'Reloj Smartwatch', 'categoria': 'Electrónica', 'precio': 199.5}, {'nombre': 'Silla Gamer', 'categoria': 'Muebles', 'precio': 199.99}, {'nombre': 'Cafetera Express', 'categoria
': 'Electrodomésticos', 'precio': 299.5}, {'nombre': 'Monitor para Computadora', 'categoria': 'Electrónica', 'precio': 349.99}, {'nombre': 'Smartphone Negro', 'categoria': 'Electrónica', 'precio': 9
99.99}]
El Tiempo final de ejecución de Merge Sort: 0.0006089210510253906
```

4 - Metodología Utilizada:

Utilizamos el lenguaje Python para desarrollar e implementar los algoritmos de búsqueda y ordenamiento utilizados. El proyecto se estructuró en los siguiente archivos principales:

- productos.py: Declara una lista que contiene 10 productos para aplicar los diferentes algoritmos de búsqueda y ordenamiento.
- buscar_nombre.py: Contiene los algoritmos de búsqueda lineal y binaria, aplicados a buscar productos por el nombre ingresado por consola.
- filtrar_categoria.py: Utiliza el algoritmo de búsqueda lineal aplicado para buscar productos que estén dentro de la categoría ingresado.
- ordenar_precio.py: Utiliza los algoritmos de ordenamiento Quick Sort y Merge Sort para ordenar los productos por precio en orden ascendente.

Las pruebas fueron realizadas a través de la consola, emulando la interacción de un usuario en una tienda online, con acciones como:

- Buscar productos por nombre
- Filtrar productos por categoría
- Ordenar productos por precios en forma ascendente.

La implementación fue acompañada de una documentación básica mediante comentarios dentro del código para facilitar su comprensión y legibilidad, permitiendo observar el funcionamiento interno de los algoritmos implementados. Además, se utilizó Git a lo largo de todo el proceso de desarrollo para llevar un control de versiones, mantener el código actualizado y resguardado en nuestro repositorio remoto

5 - Resultados Obtenidos:

Los resultados obtenidos a partir de las pruebas demostraron que las funcionalidades implementadas simulan adecuadamente los filtros de búsqueda y ordenamiento presentes en una tienda online básica. En particular:

- Filtrado por categoría: La funcionalidad retorno los resultados esperados, mostrando solo los productos correspondientes a la categoría ingresada por el usuario.
- Ordenamiento por precio: Se aplicaron exitosamente los algoritmos de Quick Sort y Merge Sort, logrando ordenar las listas de productos de manera eficiente y de forma ascendente.
- Búsqueda por nombre: tanto la búsqueda lineal como la binaria funcionaron correctamente, destacándose la velocidad de la búsqueda binaria en listas previamente ordenadas.

Las pruebas mostraron que los algoritmos funcionan correctamente con listas de tamaño medio. El ordenamiento y filtrado mejoran la organización de los datos y la búsqueda binaria resultó mucho más rápida que la lineal en listas ordenadas.

6 - Conclusión:

Este trabajo permitió aplicar de manera práctica los principales algoritmos de búsqueda y ordenamiento utilizados en plataformas de venta en línea. Se comprobó que algoritmos como Quick Sort o la búsqueda binaria no solo optimizan el rendimiento del sistema, sino que también mejoran significativamente la experiencia del usuario al facilitar el acceso a los productos de su interés y mejorando la organización de los mismos.

Además de fortalecer los conocimientos teóricos sobre algoritmos, el proyecto reforzó nuestras habilidades en programación en Python, en temas como el manejo de listas, diseño modular de código y simulación de casos reales, concluyendo que la correcta elección e implementación de estos algoritmos es clave para el desarrollo de funcionalidades robustas en plataformas digitales donde la eficiencia y escalabilidad es fundamental.

7 - Bibliografía

https://www-geeksforgeeks-org.translate.goog/linear-search-vs-binary-search/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc
https://crucialbits-com.translate.goog/blog/a-comprehensive-list-of-similarity-search-algorithms/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc
https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/
https://www.swhosting.com/es/blog/introduccion-al-algoritmo-de-ordenacion-merge-sort
https://www-geeksforgeeks-org.translate.goog/quick-sort-algorithm/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc
https://www.datacamp.com/es/tutorial/binary-search-python
https://www.datacamp.com/es/tutorial/linear-search-python