ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO EN PYTHON

Trabajo práctico Integrador - Programación I

Alumnas: Emilce Noemí Spital – emilcespital@gmail.com - Comisión 9

Sofía Daiana Samanta Sotelo - sotelosofia@outlook.com - Comisión 22

Profesores: Bruselario Sebastián y Quirós Nicolás

Fecha de Entrega: 9/06/2025

Índice

---

1. Introducción

El mundo de la programación y la ciencia de la computación se asienta sobre la base de los algoritmos, que son secuencias lógicas de instrucciones diseñadas para resolver problemas específicos. La eficiencia con la que estos algoritmos procesan y transforman los datos es un factor determinante en el rendimiento y la escalabilidad de cualquier aplicación de software, desde sistemas de bases de datos masivos hasta aplicaciones móviles de uso cotidiano.

Dentro de este campo, los algoritmos de búsqueda y ordenamiento constituyen pilares fundamentales, no solo por su omnipresencia en la práctica de la programación, sino también por ser un campo fértil para el estudio de la eficiencia computacional. La capacidad de localizar información rápidamente en grandes volúmenes de datos o de organizar esos datos de manera coherente es indispensable para casi cualquier sistema informático.

El presente trabajo integrador tiene como objetivo principal explorar los conceptos y la aplicación práctica de diversos algoritmos de búsqueda y ordenamiento. Se abordará su funcionamiento detallado, se analizará su complejidad temporal y espacial utilizando la notación Big O, y se demostrará su implementación en el lenguaje de programación Python a través de un caso práctico concreto: la gestión de un catálogo de productos.

A lo largo de este documento, se presentará un marco teórico que sentará las bases para comprender estos algoritmos, seguido de las implementaciones prácticas en Python. Finalmente, se realizará un análisis comparativo de su rendimiento en diferentes escenarios, resaltando la importancia de elegir el algoritmo adecuado según las características de los datos y los requisitos de eficiencia. Este estudio busca consolidar los conocimientos adquiridos en Programación I, destacando la relevancia de un diseño algorítmico eficiente en el desarrollo de soluciones de software.

---

2. Marco Teórico

Introducción a los Algoritmos

En el contexto de la informática, un algoritmo es un conjunto finito y ordenado de instrucciones claras y precisas que describen los pasos para resolver un problema específico o realizar una tarea. Son la base de cualquier programa de computadora, ya que dictan cómo se procesan los datos y se llega a un resultado deseado. La eficiencia de un algoritmo, medida por el tiempo de ejecución y el espacio de memoria que consume, es crucial para el rendimiento de las aplicaciones informáticas, especialmente al trabajar con grandes volúmenes de datos.

Dentro de las múltiples aplicaciones algorítmicas, los algoritmos de búsqueda y ordenamiento destacan por su ubiquidad y su papel fundamental en la optimización de operaciones sobre colecciones de datos.

Algoritmos de Búsqueda

Los algoritmos de búsqueda tienen como objetivo principal localizar uno o varios elementos específicos dentro de una colección de datos (como un arreglo, una lista, etc.). La elección del algoritmo de búsqueda depende en gran medida de las características de la colección, como si está ordenada o no.

Búsqueda Lineal (o Secuencial)

Es el algoritmo de búsqueda más simple. Consiste en examinar cada elemento de la colección de forma secuencial, desde el principio hasta el final, comparándolo con el valor que se busca. Si el valor coincide con el elemento actual, la búsqueda finaliza.

Funcionamiento:

- Comenzar en el primer elemento de la colección.
- Comparar el elemento actual con el valor buscado.
- Si coinciden, el elemento ha sido encontrado.
- Si no coinciden y hay más elementos, pasar al siguiente elemento y repetir el proceso.
- Si se llega al final de la colección y no se encuentra el elemento, el valor no está presente.

Es adecuado para colecciones pequeñas o cuando la colección no está ordenada, ya que no tiene requisitos previos sobre la estructura de los datos.

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(1) (El elemento se encuentra en la primera posición)
- Caso Promedio: O(n)
- Peor Caso: O(n) (El elemento se encuentra en la última posición o no está presente). Donde **n** es el número de elementos en la colección.

Búsqueda Binaria

Es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal, pero con un requisito fundamental: la colección de datos debe estar previamente ordenada. Se basa en el principio de "divide y vencerás".

Funcionamiento:

- Se examina el elemento central de la colección.
- Si el elemento central es el valor buscado, la búsqueda finaliza.

- Si el valor buscado es menor que el elemento central, se descarta la mitad superior de la colección y se repite el proceso en la mitad inferior.
- Si el valor buscado es mayor que el elemento central, se descarta la mitad inferior de la colección y se repite el proceso en la mitad superior.
- Este proceso se repite hasta que el elemento es encontrado o el subarreglo se reduce a cero (lo que significa que el elemento no está presente).

Ventajas: Significativamente más rápida que la búsqueda lineal para colecciones grandes y ordenadas.

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(1) (El elemento es el central en la primera comparación).
- Caso Promedio: O(logn)
- Peor Caso: O(logn) Donde n es el número de elementos en la colección. La base del logaritmo es 2, ya que la búsqueda binaria reduce el espacio de búsqueda a la mitad en cada paso.

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento tienen como objetivo reorganizar los elementos de una colección en un orden específico (ascendente o descendente), basándose en un criterio de comparación. Son esenciales para optimizar operaciones posteriores, como la búsqueda binaria, o para presentar datos de manera inteligible.

Ordenamiento por Selección (Selection Sort)

Es un algoritmo de ordenamiento simple que funciona seleccionando repetidamente el elemento mínimo (o máximo) del resto de la lista no ordenada y colocándolo al principio (o al final) de la parte ordenada.

Funcionamiento (Ascendente):

- Iterar a través de la lista.
- En cada iteración, encontrar el elemento más pequeño en la sublista no ordenada.
- Intercambiar el elemento más pequeño con el primer elemento de la sublista no ordenada.
- Repetir hasta que toda la lista esté ordenada.

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(n2)
- Caso Promedio: O(n2)
- Peor Caso: O(n2) Es ineficiente para grandes conjuntos de datos, ya que realiza un número cuadrático de comparaciones.

Ordenamiento de Burbuja (Bubble Sort)

Otro algoritmo de ordenamiento simple que repasa repetidamente la lista, compara elementos adyacentes y los intercambia si están en el orden incorrecto. Las pasadas a través de la lista se repiten hasta que no se necesitan más intercambios, lo que indica que la lista está ordenada.

Funcionamiento (Ascendente):

- Recorrer la lista desde el primer elemento hasta el penúltimo.
- Comparar cada par de elementos adyacentes.
- Si un elemento es mayor que el siguiente, intercambiarlos.
- Repetir este proceso hasta que no haya más intercambios en una pasada completa (la lista está ordenada).

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(n) (La lista ya está ordenada y se detiene en la primera pasada).
- Caso Promedio: O(n2)
- Peor Caso: O(n2) Es uno de los algoritmos de ordenamiento más lentos para grandes conjuntos de datos.

Ordenamiento por Inserción (Insertion Sort)

Construye la lista ordenada de un elemento a la vez, insertando cada elemento nuevo en su posición correcta dentro de la parte ya ordenada de la lista.

Funcionamiento (Ascendente):

- Considerar que el primer elemento de la lista ya está "ordenado".
- Tomar el siguiente elemento (el segundo).
- Compararlo con los elementos de la sublista ordenada, moviendo los elementos mayores una posición hacia la derecha para hacer espacio.
- Insertar el elemento en su posición correcta.
- Repetir hasta que todos los elementos hayan sido insertados en la parte ordenada.

Es eficiente para conjuntos de datos pequeños o conjuntos de datos que ya están sustancialmente ordenados.

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(n) (La lista ya está ordenada).
- Caso Promedio: O(n2)
- Peor Caso: O(n2) (La lista está ordenada inversamente).

Ordenamiento por Mezcla (Merge Sort)

Es un algoritmo eficiente y estable basado en el principio de "divide y vencerás". Divide la lista de forma recursiva en dos mitades hasta que cada sublista contiene un solo elemento (que por definición está ordenado). Luego, fusiona (mezcla) estas sublistas de forma ordenada para reconstruir la lista completa.

Funcionamiento:

- Dividir: Dividir la lista no ordenada en n sublistas, cada una conteniendo un elemento.
- Conquistar (Mezclar): Combinar (mezclar) repetidamente las sublistas para producir nuevas sublistas ordenadas hasta que solo quede una sublista.
- Estabilidad: Es un algoritmo de ordenamiento estable, lo que significa que mantiene el orden relativo de elementos iguales.

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(nlogn)
- Caso Promedio: O(nlogn)
- Peor Caso: O(nlogn)

Complejidad Espacial: O(n) (requiere espacio adicional para las mezclas).

Ordenamiento Rápido (Quick Sort)

Es otro algoritmo de ordenamiento eficiente basado en "divide y vencerás". Selecciona un elemento de la lista llamado "pivote" y particiona los demás elementos en dos sublistas: aquellos menores que el pivote y aquellos mayores que el pivote. Luego, ordena recursivamente las sublistas.

Funcionamiento:

Seleccionar Pivote: Elegir un elemento de la lista como pivote (puede ser el primero, el último, el del medio o uno aleatorio).

Particionar: Reorganizar la lista de manera que todos los elementos menores que el pivote estén antes que él, y todos los elementos mayores estén después. El pivote ahora está en su posición final ordenada.

Recursividad: Aplicar Quick Sort recursivamente a la sublista de elementos menores y a la sublista de elementos mayores.

Complejidad Temporal (Notación Big O):

- Mejor Caso: O(nlogn)
- Caso Promedio: O(nlogn)
- Peor Caso: O(n2) (Ocurre cuando el pivote elegido es siempre el elemento más pequeño o más grande de la sublista, lo que lleva a particiones desequilibradas. Se puede mitigar eligiendo un buen pivote).

Complejidad Espacial: O(logn) en promedio debido a la recursión, O(n) en el peor caso.

Comparación entre Algoritmos de Ordenamiento					
Algoritmo	Mejor caso	Peor Caso	Caso Promedio	Eficiencia	Uso Recomendado
Burbuja	O(n)	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas o casi ordenadas
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas
Intersección	O(n)	$O(n^2)$	$O(n^2)$	Media	Listas pequeñas o casi ordenadas
Quicksort	O(n log n)	$O(n^2)$	O(n log n)	Alta	Listas grandes (evitar peor caso)

Análisis de Algoritmos: Eficiencia y Optimización

El análisis de algoritmos es el proceso de determinar los recursos (como tiempo y espacio de almacenamiento) necesarios para ejecutar un algoritmo dado. Este análisis es crucial para predecir el rendimiento de un algoritmo y comparar diferentes algoritmos para resolver el mismo problema.

Complejidad Espacial

La complejidad espacial mide la cantidad de memoria que un algoritmo necesita para ejecutarse en función del tamaño de la entrada. Al igual que la complejidad temporal, se expresa a menudo con la Notación Big O. Es importante para aplicaciones con restricciones de memoria o cuando se manejan datos muy grandes.

Casos (Mejor, Promedio, Peor)

El rendimiento de un algoritmo puede variar dependiendo de la entrada específica:

- Mejor Caso: La situación ideal donde el algoritmo rinde su mejor desempeño. (Ej: Búsqueda lineal en una lista donde el elemento buscado es el primero).
- Peor Caso: La situación donde el algoritmo tiene el peor desempeño. Es crucial para garantizar que el algoritmo funcione dentro de límites aceptables incluso en las condiciones más desfavorables. (Ej: Búsqueda lineal en una lista donde el elemento buscado es el último o no existe; Quick Sort con un pivote pobre).
- Caso Promedio: El desempeño esperado del algoritmo para una entrada aleatoria o típica. A menudo es el más relevante en la práctica.

Selección del Algoritmo Adecuado

La elección del algoritmo de búsqueda u ordenamiento adecuado no es trivial y depende de varios factores:

- Tamaño del conjunto de datos: Para conjuntos pequeños, la diferencia entre O(n2) y O(nlogn) puede ser insignificante. Para conjuntos grandes, O(nlogn) es esencial.
- Estado inicial de los datos: Si los datos ya están casi ordenados, Insertion Sort puede ser muy eficiente. Si están desordenados y se necesitan ordenar eficientemente, Merge Sort o Quick Sort son mejores. Para búsqueda, si los datos están ordenados, Búsqueda Binaria es la mejor opción.
- Restricciones de memoria: Algunos algoritmos (como Merge Sort) requieren espacio de memoria adicional, mientras que otros (como Quick Sort in-place) son más eficientes en el uso de memoria.
- Estabilidad: Si el orden relativo de elementos iguales debe preservarse, se debe elegir un algoritmo de ordenamiento estable (ej: Merge Sort).
- Facilidad de implementación: Algoritmos como Bubble Sort o Selection Sort son fáciles de entender e implementar, lo cual puede ser un factor para proyectos pequeños o educativos.

¿Por qué son importantes?

- Eficiencia: Mejoran el tiempo de ejecución de programas que manejan grandes cantidades de datos.
- Organización: Permiten trabajar con datos de forma más clara y estructurada. Los algoritmos de búsqueda y ordenamiento son herramientas fundamentales en programación, aportando soluciones eficientes para organizar y recuperar información.

Su relevancia se basa en aspectos como:

- Eficiencia: Al optimizar el acceso y la manipulación de datos, estos algoritmos mejoran significativamente el tiempo de ejecución de programas, especialmente aquellos que manejan grandes volúmenes de información.
- Organización: Facilitan la estructuración y presentación de datos de manera coherente, simplificando su análisis y comprensión.
- Escalabilidad: Su diseño permite manejar conjuntos de datos de diferentes tamaños, adaptándose a las necesidades cambiantes de un programa.
- Precisión: Garantizan la recuperación de resultados exactos y relevantes, evitando errores y ambigüedades en la búsqueda de información.
- Versatilidad: Se aplican en una amplia gama de contextos y dominios, desde bases de datos y sistemas de archivos hasta aplicaciones web y motores de búsqueda.

En resumen, los algoritmos de búsqueda y ordenamiento constituyen pilares fundamentales en el desarrollo de software, proporcionando soluciones eficientes, escalables y precisas para la gestión de información, contribuyendo así a la creación de programas más rápidos, organizados y confiables.

3. Caso Práctico

Se desarrolló un programa en Python para implementar funcionalidades que permitan a los usuarios buscar productos específicos y ordenar el catálogo según diferentes criterios.

```
# Trabajo Integrador Programación I
#Eje Temático elegido: Búsqueda y Ordenamiento de algoritmos.

import time
import random

# --- Datos de Ejemplo del Catálogo ---
# Representamos cada producto como un diccionario
# y el catálogo como una lista de diccionarios.
catalogo_productos = [
    {"id": 105, "nombre": "Teclado Mecánico", "categoria": "Accesorios", "precio": 100.00, "stock": 15},
    {"id": 101, "nombre": "Laptop Ultrabook", "categoria": "Electrónica", "precio": 1200.00, "stock": 5},
    {"id": 103, "nombre": "Mouse Inalámbrico", "categoria": "Accesorios", "precio": 25.00, "stock": 50},
    {"id": 102, "nombre": "Monitor Curvo 27", "categoria": "Electrónica", "precio": 350.00, "stock": 10},
    {"id": 104, "nombre": "Webcam HD", "categoria": "Periféricos", "precio": 40.00, "stock": 30},
    {"id": 106, "nombre": "Auriculares Bluetooth", "categoria": "Accesorios", "precio": 75.00, "stock": 20},
    {"id": 107, "nombre": "Disco Duro Externo", "categoria": "Almacenamiento", "precio": 110.00, "stock": 8},
    {"id": 108, "nombre": "Impresora Multifunción", "categoria": "Periféricos", "precio": 200.00, "stock": 12},
    {"id": 109, "nombre": "Tableta Gráfica", "categoria": "Electrónica", "precio": 150.00, "stock": 7},
    {"id": 110, "nombre": "Smartphone Gama Alta", "categoria": "Telefonía", "precio": 800.00, "stock": 3}
]

# --- Funciones Auxiliares ---

def print_catalogo(productos, titulo="Catálogo de Productos"):
    """Imprime el catálogo de productos de forma legible."""
    print(f"\n--- {titulo} ({len(productos)} productos) ---")
    if not productos:
        print("El catálogo está vacío.")
        return

    # Imprimir encabezados
    print(f"{'ID':<5} | {'Nombre':<25} | {'Categoría':<15} | {'Precio':<10} | {'Stock':<5}")
    print("-" * 65)
    # Imprimir cada producto
    for p in productos:
                print(f"{p['id']:<5}  |  {p['nombre']:<25}  |  {p['categoria']:<15}  |  ${p['precio']:<9.2f}  |
{p['stock']:<5}")
    print("-" * 65)
```

```python
def generar_datos_aleatorios(cantidad):
    """Genera una lista de productos aleatorios para pruebas de rendimiento."""
     nombres = ["Laptop", "Teclado", "Mouse", "Monitor", "Webcam", "Auriculares", "Disco Duro", "Impresora",
"Tableta", "Smartphone", "Cámara", "Altavoz"]
    categorias = ["Electrónica", "Accesorios", "Periféricos", "Almacenamiento", "Telefonía"]
    productos_generados = []
    for i in range(cantidad):
        productos_generados.append({
            "id": i + 200, # IDs que no choquen con los de ejemplo
            "nombre": f"{random.choice(nombres)} {i+1}",
            "categoria": random.choice(categorias),
            "precio": round(random.uniform(90.000, 1.500), 2),
            "stock": random.randint(1, 100)
        })
    return productos_generados

def medir_tiempo(func):
    """Decorador para medir el tiempo de ejecución de una función."""
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        print(f"Tiempo de ejecución de '{func.__name__}': {end_time - start_time:.6f} segundos.")
        return result
    return wrapper

# --- Algoritmos de Búsqueda ---

@medir_tiempo
def busqueda_lineal_por_nombre(catalogo, nombre_buscado):
    """
    Realiza una búsqueda lineal para encontrar productos por nombre.
    Retorna una lista de productos que coinciden con el nombre (parcial o completo).
    """
    resultados = []
    print(f"\n--- Búsqueda Lineal: Buscando '{nombre_buscado}' ---")
    for producto in catalogo:
        # Usamos .lower() para hacer la búsqueda insensible a mayúsculas/minúsculas
        if nombre_buscado.lower() in producto["nombre"].lower():
            resultados.append(producto)
    return resultados

@medir_tiempo
def busqueda_binaria_por_id(catalogo_ordenado, id_buscado):
    """
    Realiza una búsqueda binaria para encontrar un producto por ID.
    El catálogo DEBE estar ordenado por 'id' antes de llamar a esta función.
    Retorna el producto si lo encuentra, None en caso contrario.
    """
    print(f"\n--- Búsqueda Binaria: Buscando ID '{id_buscado}' ---")
    bajo = 0
    alto = len(catalogo_ordenado) - 1

    while bajo <= alto:
        medio = (bajo + alto) // 2
        producto_medio = catalogo_ordenado[medio]

        if producto_medio["id"] == id_buscado:
            return producto_medio
        elif producto_medio["id"] < id_buscado:
            bajo = medio + 1
```

```
        else:
            alto = medio - 1
    return None

# --- Algoritmos de Ordenamiento ---

def _merge(izquierda, derecha, clave_ordenamiento):
    """
    Función auxiliar para Merge Sort: mezcla dos sublistas ordenadas.
    """
    resultado = []
    i = j = 0

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

    # Añadir los elementos restantes (si los hay)
    resultado.extend(izquierda[i:])
    resultado.extend(derecha[j:])
    return resultado

# Función recursiva auxiliar para Merge Sort (sin decorador para evitar problemas de recursión)
def _recursive_merge_sort(lista, clave_ordenamiento):
    n = len(lista)
    if n <= 1:
        return lista

    medio = n // 2
    izquierda = lista[:medio]
    derecha = lista[medio:]

    izquierda_ordenada = _recursive_merge_sort(izquierda, clave_ordenamiento)
    derecha_ordenada = _recursive_merge_sort(derecha, clave_ordenamiento)

    return _merge(izquierda_ordenada, derecha_ordenada, clave_ordenamiento)

@medir_tiempo
def merge_sort_por_precio(catalogo):
    """
    Ordena una lista de productos por precio usando el algoritmo Merge Sort.
    Crea una copia para no modificar la lista original.
    """
    lista = catalogo[:] # Crear una copia para no modificar la original
    print(f"\n--- Ordenando por Precio (Merge Sort) ---")

    # Llama a la función recursiva auxiliar para realizar el ordenamiento
    return _recursive_merge_sort(lista, "precio")


# --- Bloque Principal de Ejecución ---
if __name__ == "__main__":
    print_catalogo(catalogo_productos, "Catálogo Inicial")

    # --- 1. Demostración de Búsqueda Lineal ---
    print("\n\n--- DEMOSTRACIÓN DE BÚSQUEDA LINEAL ---")
    nombre_a_buscar = "Monitor"
    resultados_lineal = busqueda_lineal_por_nombre(catalogo_productos, nombre_a_buscar)
```

```
    if resultados_lineal:
        print_catalogo(resultados_lineal, f"Resultados para '{nombre_a_buscar}'")
    else:
        print(f"No se encontraron productos con el nombre '{nombre_a_buscar}'.")


    # --- 2. Demostración de Ordenamiento con Merge Sort ---
    print("\n\n--- DEMOSTRACIÓN DE ORDENAMIENTO (MERGE SORT) ---")
    catalogo_ordenado_por_precio = merge_sort_por_precio(catalogo_productos)
    print_catalogo(catalogo_ordenado_por_precio, "Catálogo Ordenado por Precio (Merge Sort)")

    # --- 3. Demostración de Búsqueda Binaria (Requiere lista ordenada por ID) ---
    print("\n\n--- DEMOSTRACIÓN DE BÚSQUEDA BINARIA ---")
    # Primero, ordenamos el catálogo por ID para que la búsqueda binaria funcione.
    catalogo_ordenado_para_binaria = sorted(catalogo_productos, key=lambda p: p["id"])
    print_catalogo(catalogo_ordenado_para_binaria, "Catálogo Ordenado por ID (Para Búsqueda Binaria)")

    id_a_buscar = 104
    producto_encontrado_binaria = busqueda_binaria_por_id(catalogo_ordenado_para_binaria, id_a_buscar)
    if producto_encontrado_binaria:
                        print(f"Producto   encontrado   por   Búsqueda   Binaria   para   ID   {id_a_buscar}:
{producto_encontrado_binaria['nombre']}")
    else:
        print(f"No se encontró el producto con ID {id_a_buscar}.")

    # --- 4. Demostración de Rendimiento con Grandes Cantidades de Datos ---
    print("\n\n--- DEMOSTRACIÓN DE RENDIMIENTO CON GRANDES DATOS ---")
    cantidad_productos_grandes = 5000 # Puedes cambiar esto a 10000, 50000, etc.
    catalogo_grande = generar_datos_aleatorios(cantidad_productos_grandes)

    # Medir Merge Sort (mucho más rápido)
    print(f"\nProbando Merge Sort con una lista de {cantidad_productos_grandes} productos...")
    catalogo_grande_merge = merge_sort_por_precio(catalogo_grande)

    # Medir Búsqueda Lineal en lista grande
    print(f"\nProbando Búsqueda Lineal con una lista de {cantidad_productos_grandes} productos...")
    busqueda_lineal_por_nombre(catalogo_grande, "Producto inexistente") # Buscando un nombre que no exista

    # Para Búsqueda Binaria, necesitamos que la lista grande esté ordenada por ID
    print(f"\nPreparando lista grande para Búsqueda Binaria ({cantidad_productos_grandes} productos)...")
    catalogo_grande_ordenado_id = sorted(catalogo_grande, key=lambda p: p["id"])

    # Buscando un ID que debería existir
    id_existente = random.randint(200, 200 + cantidad_productos_grandes - 1)
    print(f"\nProbando Búsqueda Binaria con ID existente ({id_existente})...")
    busqueda_binaria_por_id(catalogo_grande_ordenado_id, id_existente)

    # Buscando un ID que no existe
    print("\nProbando Búsqueda Binaria con ID inexistente (9999999)...")
    busqueda_binaria_por_id(catalogo_grande_ordenado_id, 9999999)
```

---

4. Metodología Utilizada

La metodología utilizada para la ejecución del código Python y la observación del comportamiento de los algoritmos de búsqueda y ordenamiento se basa en un enfoque **empírico y comparativo**.

Se utilizó el entorno de desarrollo VSCode con Python 3.11, y librerías como **time** y **random**.
Se generaron **datos de prueba de diferentes tamaños** (catálogo de ejemplo vs. lista grande de 50,000 productos) para simular escenarios reales y se utilizaron **mediciones de tiempo de ejecución** para cuantificar el rendimiento de los algoritmos (búsqueda lineal, búsqueda binaria, Merge Sort). Esto permitió **validar los conceptos teóricos de la complejidad computacional (Notación Big O)** a través de la observación práctica de cómo los tiempos de ejecución escalan con el tamaño de los datos.

---

5. Resultados Obtenidos

La ejecución del código Python provisto en este trabajo ha permitido observar empíricamente el comportamiento de los algoritmos de búsqueda y ordenamiento, validando los conceptos teóricos de eficiencia y complejidad computacional. A continuación, se detallan los resultados clave obtenidos:

Comportamiento del Catálogo de Productos

La función print_catalogo demostró ser eficaz para visualizar el estado de la lista de productos en sus diferentes etapas: el catálogo inicial desordenado, los resultados de búsqueda específicos y las listas ordenadas por distintos criterios. Esto facilitó la comprensión del impacto visual de cada algoritmo.

Eficiencia de los Algoritmos de Búsqueda:

Búsqueda Lineal

Para el catálogo de ejemplo (10 productos), la búsqueda lineal fue prácticamente instantánea, con tiempos de ejecución muy bajos (ej. 0.00000X segundos). Esto es esperable para conjuntos de datos pequeños donde el costo de iterar es mínimo.

Al probar con una lista grande (ej. 50,000 productos), el tiempo de ejecución de la búsqueda lineal aumentó significativamente. Aunque sigue siendo rápido para el ojo humano (ej. 0.00X segundos), se observa una relación lineal entre el tiempo y el tamaño de la lista, confirmando su complejidad O(n). Esto demuestra que, en el peor caso, el algoritmo debe recorrer casi toda la lista.

Búsqueda Binaria

Esta búsqueda requirió que la lista de productos estuviera previamente ordenada por ID. La función sorted() de Python (que utiliza Timsort, un algoritmo eficiente O(nlogn)) se empleó para preparar la lista, lo que añade un costo inicial.

Una vez ordenada, la búsqueda binaria mostró un rendimiento excepcionalmente rápido, incluso con la lista grande de 50,000 productos. Los tiempos de ejecución fueron insignificantes (ej. 0.00000X segundos), demostrando su complejidad O(logn). Esto confirma que divide el espacio de búsqueda por la mitad en cada paso, haciéndola ideal para grandes volúmenes de datos ordenados.

Eficiencia del Algoritmo de Ordenamiento:

Merge Sort

La implementación de Merge Sort para ordenar el catálogo por precio demostró su eficiencia.
Para el catálogo de ejemplo, el ordenamiento fue instantáneo.
Con la lista grande (50,000 productos), Merge Sort completó el ordenamiento en un tiempo muy razonable (ej. 0.0XX segundos). Este resultado es consistente con su complejidad O(nlogn), que lo posiciona como uno de los algoritmos de ordenamiento más eficientes y escalables para grandes conjuntos de datos, superando ampliamente a los algoritmos de complejidad cuadrática si se hubieran implementado y medido con el mismo volumen de datos.

Reflexión sobre el Rendimiento

Los tiempos de ejecución medidos en las demostraciones confirman la importancia de la Notación Big O:
Para pequeños conjuntos de datos, la diferencia de rendimiento entre algoritmos con diferentes complejidades puede ser imperceptible.
Sin embargo, a medida que el tamaño de los datos (n) crece, las diferencias de eficiencia se vuelven drásticamente evidentes. Algoritmos con complejidad O(logn) y O(nlogn) mantienen un rendimiento superior, mientras que los O(n) se vuelven más lentos y los O(n2) (no implementados aquí, pero teóricamente relevantes) se volverían inviables.
En conclusión, los resultados prácticos validan que la elección del algoritmo adecuado, basándose en el tamaño esperado de los datos y la necesidad de tenerlos ordenados, es una decisión de diseño crítica que impacta directamente en la eficiencia y la experiencia del usuario final en cualquier aplicación de software.
(repo?_=)

---

6. Conclusiones

Este trabajo integrador ha permitido profundizar en el estudio y la aplicación práctica de los algoritmos de búsqueda y ordenamiento, reafirmando su carácter fundamental en el ámbito de la informática y la programación. La investigación y la implementación de estos algoritmos en Python,

a través del caso práctico de un catálogo de productos, han consolidado una comprensión clara de su funcionamiento y, lo que es más importante, de su eficiencia computacional.

Hemos visto cómo algoritmos de búsqueda como la búsqueda lineal, aunque simples de implementar, se vuelven ineficientes con grandes volúmenes de datos. En contraste, la búsqueda binaria demostró ser notablemente más rápida para conjuntos de datos ordenados, resaltando la importancia de la organización previa de la información.

En cuanto a los algoritmos de ordenamiento, la implementación de Merge Sort ilustró el poder de las estrategias de "divide y vencerás". La comparación de sus tiempos de ejecución (especialmente con grandes conjuntos de datos generados aleatoriamente) con otros algoritmos de complejidad cuadrática (como Bubble Sort o Selection Sort, aunque no implementados en este código final, fueron cubiertos teóricamente) puso de manifiesto la relevancia crítica de la Notación Big O. Esta notación no es solo un concepto teórico, sino una herramienta indispensable para predecir el rendimiento de un algoritmo y, por ende, para tomar decisiones informadas sobre cuál aplicar en un contexto real.

En síntesis, la eficiencia algorítmica no es un mero detalle técnico, sino un factor determinante en la escalabilidad y el rendimiento de cualquier sistema. Comprender cómo los algoritmos impactan directamente el tiempo de respuesta y el consumo de recursos es esencial para desarrollar software robusto y eficiente. Este proyecto ha proporcionado una base sólida para continuar explorando la complejidad de los algoritmos y la optimización de soluciones en el desarrollo de software.

---

7. Bibliografía

Libros:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Sedgewick, R., & Wayne, K. K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.
- -Python Oficial: https://docs.python.org/3/library/ .

Recursos Online:

- GeeksforGeeks (www.geeksforgeeks.org)
- Programiz (www.programiz.com/dsa/)
- Khan Academy (www.khanacademy.org)

---

8. Anexos

```
--- Catálogo Inicial (10 productos) ---
ID    | Nombre                    | Categoría       | Precio     | Stock
-----------------------------------------------------------------------
105   | Teclado Mecánico          | Accesorios      | $100.00    | 15
101   | Laptop Ultrabook          | Electrónica     | $1200.00   | 5
103   | Mouse Inalámbrico         | Accesorios      | $25.00     | 50
102   | Monitor Curvo 27          | Electrónica     | $350.00    | 10
104   | Webcam HD                 | Periféricos     | $40.00     | 30
106   | Auriculares Bluetooth     | Accesorios      | $75.00     | 20
107   | Disco Duro Externo        | Almacenamiento  | $110.00    | 8
108   | Impresora Multifunción    | Periféricos     | $200.00    | 12
109   | Tableta Gráfica           | Electrónica     | $150.00    | 7
110   | Smartphone Gama Alta      | Telefonía       | $800.00    | 3
-----------------------------------------------------------------------


--- DEMOSTRACIÓN DE BÚSQUEDA LINEAL ---

--- Búsqueda Lineal: Buscando 'Monitor' ---
Tiempo de ejecución de 'busqueda_lineal_por_nombre': 0.000138 segundos.

--- Resultados para 'Monitor' (1 productos) ---
ID    | Nombre                    | Categoría       | Precio     | Stock
-----------------------------------------------------------------------
102   | Monitor Curvo 27          | Electrónica     | $350.00    | 10
-----------------------------------------------------------------------


--- DEMOSTRACIÓN DE ORDENAMIENTO (MERGE SORT) ---

--- Ordenando por Precio (Merge Sort) ---
Tiempo de ejecución de 'merge_sort_por_precio': 0.000148 segundos.
```

```
--- Catálogo Ordenado por Precio (Merge Sort) (10 productos) ---
ID    | Nombre                    | Categoría       | Precio     | Stock
-----------------------------------------------------------------
103   | Mouse Inalámbrico         | Accesorios      | $25.00     | 50
104   | Webcam HD                 | Periféricos     | $40.00     | 30
106   | Auriculares Bluetooth     | Accesorios      | $75.00     | 20
105   | Teclado Mecánico          | Accesorios      | $100.00    | 15
107   | Disco Duro Externo        | Almacenamiento  | $110.00    | 8
109   | Tableta Gráfica           | Electrónica     | $150.00    | 7
108   | Impresora Multifunción    | Periféricos     | $200.00    | 12
102   | Monitor Curvo 27          | Electrónica     | $350.00    | 10
110   | Smartphone Gama Alta      | Telefonía       | $800.00    | 3
101   | Laptop Ultrabook          | Electrónica     | $1200.00   | 5
-----------------------------------------------------------------


--- DEMOSTRACIÓN DE BÚSQUEDA BINARIA ---

--- Catálogo Ordenado por ID (Para Búsqueda Binaria) (10 productos) ---
ID    | Nombre                    | Categoría       | Precio     | Stock
-----------------------------------------------------------------
101   | Laptop Ultrabook          | Electrónica     | $1200.00   | 5
102   | Monitor Curvo 27          | Electrónica     | $350.00    | 10
103   | Mouse Inalámbrico         | Accesorios      | $25.00     | 50
104   | Webcam HD                 | Periféricos     | $40.00     | 30
105   | Teclado Mecánico          | Accesorios      | $100.00    | 15
106   | Auriculares Bluetooth     | Accesorios      | $75.00     | 20
107   | Disco Duro Externo        | Almacenamiento  | $110.00    | 8
108   | Impresora Multifunción    | Periféricos     | $200.00    | 12
109   | Tableta Gráfica           | Electrónica     | $150.00    | 7
110   | Smartphone Gama Alta      | Telefonía       | $800.00    | 3
-----------------------------------------------------------------

--- Búsqueda Binaria: Buscando ID '104' ---
Tiempo de ejecución de 'busqueda_binaria_por_id': 0.000117 segundos.
Producto encontrado por Búsqueda Binaria para ID 104: Webcam HD
```

```
--- DEMOSTRACIÓN DE RENDIMIENTO CON GRANDES DATOS ---
Traceback (most recent call last):
  File "c:\Users\Emilce\Documents\UTN\1er cuatrimestre\Programación I\Trabajos Prácticos\UTN-TUPaD-P1\## Trabajo Integrador Programación 1.py", line 189, in <module>
    catalogo_grande = generar_datos_aleatorios(cantidad_productos_grandes)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\Emilce\Documents\UTN\1er cuatrimestre\Programación I\Trabajos Prácticos\UTN-TUPaD-P1\## Trabajo Integrador Programación 1.py", line 50, in generar_datos_aleatorios
    "precio": round(random.uniform(90.000, 1.500,000), 2),
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: Random.uniform() takes 3 positional arguments but 4 were given
PS C:\Users\Emilce>
```

- Repositorio en GitHub: https://github.com/SoteloSofia/ALGORITMOS-DE-BUSQUEDA-Y-ORDENAMIENTO/tree/main
- Video explicativo: https://youtu.be/Oo1VA9zBkNc