Trabajo Integrador – Programación I

Algoritmos de Búsqueda y Ordenamiento en Python

Diego Velardes – Agustín Spinotti

Profesor: Nicolás Quiros

Fecha: 08/06/2025

Contenido

Introducción

En el vasto universo de la informática, la eficiencia en el manejo de datos es una piedra angular para el desarrollo de cualquier aplicación robusta y funcional. Ya sea organizando una base de datos de clientes, optimizando una ruta de entrega, o gestionando el stock de un almacén, la capacidad de buscar información rápidamente y ordenarla de manera lógica es fundamental. Este Trabajo Práctico se sumerge en el estudio y la aplicación de algoritmos clave de búsqueda y ordenamiento, pilares de la programación y la ciencia de datos.

El objetivo de este proyecto es demostrar y comparar el rendimiento de distintos algoritmos de ordenamiento y búsqueda, utilizando como caso práctico un sistema de gestión de inventario simplificado. Nos enfocaremos en la manipulación de una colección de productos, cada uno definido por un ID, nombre, categoría y precio. A través de la implementación de algoritmos como Bubble Sort, Insertion Sort, y Merge Sort para el ordenamiento de productos por precio, y la búsqueda lineal junto a la búsqueda binaria para la localización de productos por ID, se buscará evidenciar cómo la elección del algoritmo correcto impacta drásticamente en la eficiencia operativa, especialmente a medida que el volumen de datos se incrementa.

A lo largo de este documento, se detallarán las implementaciones de cada algoritmo, se presentarán los resultados de las pruebas de tiempo de ejecución realizadas con diferentes tamaños de inventario, y se ofrecerá un análisis comparativo que justifique las ventajas de unos sobre otros en escenarios específicos. Este enfoque práctico nos permitirá comprender no solo el funcionamiento interno de estos algoritmos, sino también su relevancia directa en la optimización de tareas cotidianas en el ámbito de la gestión de datos.

Marco Teórico

Cuando hablamos de algoritmos informáticos, nos referimos a la fase crucial previa a la escritura de cualquier programa. Antes de que un desarrollador empiece a "hablar" con la máquina a través del código, primero debe diseñar la solución al problema en forma de algoritmo. Es como trazar un plano detallado antes de construir un edificio.

En este contexto, un algoritmo informático describe:

- Datos de entrada: La información inicial que el algoritmo necesita para empezar a trabajar.
- Operaciones a realizar: Los pasos lógicos y matemáticos que el algoritmo debe llevar a cabo.
- Datos de salida: El resultado o la solución que el algoritmo produce.

Una vez que el algoritmo está definido, el siguiente paso es codificarlo en un lenguaje de programación (como Python). Este código es la traducción del algoritmo a un idioma que la computadora puede entender y ejecutar. Por lo tanto, un programa informático no es más que una colección de algoritmos bien estructurados y traducidos a un lenguaje que una máquina puede procesar para cumplir una función específica.

La búsqueda es el proceso de encontrar uno o más elementos específicos dentro de una colección de datos que satisfacen ciertas condiciones. El algoritmo de búsqueda binaria es un ejemplo claro de la técnica Divide y Vencerás. El problema de partida es decidir si existe un elemento dado x en un vector de enteros ordenado. El hecho de que esté ordenado va a permitir utilizar esta técnica, pues podemos plantear un algoritmo con la siguiente estrategia: compárese el elemento dado x con el que ocupa la posición central del vector. En caso de que coincida con él, hemos solucionado el problema. Pero si son distintos, pueden darse dos situaciones: que x sea mayor que el elemento en posición central, o que sea menor. En cualquiera de los dos casos podemos descartar una de las dos mitades del vector, puesto que, si x es mayor que el elemento en posición central, también será mayor que todos los elementos en

posiciones anteriores, y al revés. Ahora se procede de forma recursiva sobre la mitad que no hemos descartado. En este ejemplo la división del problema es fácil, puesto que en cada paso se divide el vector en dos mitades tomando como referencia su posición central. El problema queda reducido a uno de menor tamaño y por ello hablamos de divide y vencerás.

Su caso base se produce cuando el vector tiene sólo un elemento. En esta situación la solución del problema se basa en comparar dicho elemento con x. Como el tamaño de la entrada (en este caso el número de elementos del vector a tratar) se va dividiendo en cada paso por dos, tenemos asegurada la convergencia al caso base.

La búsqueda lineal consiste en recorrer una lista desde el inicio hasta el final comparando uno por uno cada elemento. Es sencilla de implementar y funciona incluso si la lista no está ordenada, pero su eficiencia es baja en grandes volúmenes de datos (O(n)).

La búsqueda binaria, en cambio, requiere que la lista esté ordenada. Divide el espacio de búsqueda en mitades sucesivas, reduciendo el número de comparaciones necesarias. Su eficiencia es O(log n), siendo ideal para estructuras grandes siempre que se mantenga el orden de los datos.

Ambos algoritmos permiten encontrar elementos, pero con eficiencias muy distintas.

```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    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
```

Desde los inicios de la computación, los algoritmos de ordenamiento han captado un gran interés e impulsado una constante investigación. Aunque la tarea de ordenar una lista puede parecer sencilla a primera vista, a lo largo de la historia se han desarrollado incontables técnicas para lograr la forma más rápida y eficiente de hacerlo. Esta búsqueda incesante por la optimización convierte a los algoritmos de ordenamiento en un campo fascinante y fundamental.

La vasta diversidad de estos algoritmos los establece como un tema excelente para el aprendizaje práctico de cualquier lenguaje de programación. Al adentrarse en su implementación, se aplican y consolidan conceptos esenciales como el manejo de arreglos, la lógica de operaciones de comparación y el uso estratégico de operaciones aritméticas.

El ordenamiento consiste en organizar un conjunto de elementos en una secuencia determinada, basándose en un criterio específico (numérico, alfabético, cronológico, etc.). La eficiencia de un algoritmo de ordenamiento se mide por su complejidad temporal, que describe cómo el tiempo de ejecución crece a medida que el tamaño de la entrada (N) aumenta. Esta complejidad se expresa comúnmente usando la notación Big O (O), que representa el límite superior del tiempo de ejecución en el peor de los casos.

La complejidad algorítmica es el estudio de la eficiencia de los algoritmos en términos del tiempo y el espacio computacional que requieren. Comprender la notación Big O es crucial porque nos permite predecir cómo se comportará un algoritmo a medida que el tamaño de la entrada crece.

- O(N2) (Cuadrático): Algoritmos que caen en esta categoría (como Bubble Sort e Insertion Sort) se vuelven muy lentos muy rápidamente a medida que N aumenta. No son adecuados para grandes volúmenes de datos.
- O(NlogN) (Logarítmico-Lineal): Algoritmos como Merge Sort y Timsort exhiben un crecimiento mucho más manejable. Son considerados eficientes y escalables para la mayoría de las aplicaciones.
- O(logN) (Logarítmico): Algoritmos como la búsqueda binaria son extremadamente eficientes. El tiempo de ejecución apenas aumenta, incluso con incrementos masivos en N.
- O(N) (Lineal): Algoritmos como la búsqueda lineal tienen un crecimiento directamente proporcional a N. Son aceptables para listas pequeñas o cuando no se puede garantizar el orden.

En el presente desarrollaremos todos ellos con detenimiento, prestando especial atención a su complejidad, no sólo en el caso medio sino también en los casos mejor y peor, pues para algunos existen diferencias significativas.

Bubble Sort Es uno de los algoritmos de ordenamiento más simples. funciona comparando repetidamente pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que no se necesitan más intercambios, indicando que la lista está ordenada. Es como las "burbujas" más pesadas se hunden y las ligeras flotan. Su complejidad está dada por O(N2) en el peor y caso promedio. Esto significa que, si el número de elementos se duplica, el tiempo de ejecución se cuadruplica. En el mejor caso (lista ya ordenada), es O(N), pero esta situación es rara en la práctica.

```
def bubble_sort(lista):
    for i in range(len(lista)):
        for j in range(0, len(lista) - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

Insertion Sort este algoritmo construye la lista ordenada un elemento a la vez. Toma cada elemento de la parte no ordenada de la lista y lo inserta en su posición correcta dentro de la parte ya ordenada de la lista. Es análogo a cómo

una persona ordena un mazo de cartas en su mano. Su complejidad esta dada por O(N2) en el peor y caso promedio. Su rendimiento también se degrada cuadráticamente con el aumento de N. En el mejor caso (lista ya ordenada), es O(N), lo que lo hace más eficiente que Bubble Sort en esa situación.

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

Merge Sort es un algoritmo de ordenamiento eficiente basado en el paradigma "Divide y Vencerás". Funciona recursivamente dividiendo la lista en mitades hasta que cada sublista contiene un solo elemento (que, por definición, está ordenado). Luego, estas sublistas ordenadas se combinan (fusionan) de manera que el resultado es una lista más grande y ordenada. Su complejidad es O(NlogN) en todos los casos (mejor, promedio y peor). Esta complejidad logarítmica lineal lo hace muy eficiente para grandes conjuntos de datos, ya que el crecimiento del tiempo es mucho más lento que el cuadrático.

```
def merge_sort(lista):
    if len(lista) <= 1:
        return lista
    medio = len(lista) // 2
    izquierda = merge_sort(lista[:medio])
    derecha = merge_sort(lista[medio:])
    return merge(izquierda, derecha)


def merge(izq, der):
    resultado = []
    i = j = 0
    while i < len(izq) and j < len(der):
```

```
        if izq[i] < der[j]:
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1
    resultado += izq[i:]
    resultado += der[j:]
    return resultado
```

Python Timsort si bien no es un algoritmo implementado manualmente en nuestro TP, es crucial mencionarlo ya que es el algoritmo de ordenamiento por defecto en Python. Timsort es un algoritmo de ordenamiento híbrido, una combinación de Merge Sort e Insertion Sort. Aprovecha las ventajas de Insertion Sort para pequeñas sublistas (donde es muy eficiente) y de Merge Sort para combinar esas sublistas más grandes.

Caso Practico

Este script Python ha sido diseñado para demostrar y comparar el rendimiento de diferentes algoritmos de ordenamiento y búsqueda en un contexto simplificado de gestión de inventario. Su objetivo principal es medir los tiempos de ejecución, permitiendo una clara visualización de las diferencias de eficiencia entre los distintos enfoques algorítmicos.

El código comienza definiendo una estructura de datos simplificada para el inventario. Cada producto se representa como un diccionario de Python con cuatro claves principales: id_producto, nombre, categoria y precio. Para generar datos de prueba de forma dinámica, se utiliza la función generar_inventario_aleatorio(cantidad). Esta función crea una lista de estos diccionarios, poblándolos con IDs secuenciales y asignando nombres, categorías y precios aleatorios, utilizando un diccionario predefinido (DATOS_PRODUCTOS_EJEMPLO) para simular un conjunto de datos más

realista. La capacidad de generar inventarios de diferentes tamaños (pequeño, mediano, grande) es crucial para observar cómo el rendimiento de los algoritmos escala con el volumen de datos.

```
def generar_inventario_aleatorio(cantidad):
    productos_generados = []
    categorias = list(DATOS_PRODUCTOS_EJEMPLO.keys())
    for i in range(cantidad):
        id_prod = f"PROD{i:05d}"
        categoria_prod = random.choice(categorias)
        nombre_prod =
random.choice(DATOS_PRODUCTOS_EJEMPLO[categoria_prod]) +
f"_{i}"
        precio_prod = round(random.uniform(1.0, 500.0), 2)
        productos_generados.append({
            "id_producto": id_prod,
            "nombre": nombre_prod,
            "categoria": categoria_prod,
            "precio": precio_prod
        })
    return productos_generados
```

Para la funcionalidad de búsqueda, el script incluye dos métodos principales:

Búsqueda Lineal (busqueda_lineal_por_id): Esta función implementa el algoritmo de búsqueda lineal básico. Recorre la lista de productos elemento por elemento, comparando el id_producto de cada uno con el id_buscar proporcionado. Si encuentra una coincidencia, devuelve el producto; de lo contrario, si recorre toda la lista sin éxito, retorna None. Es un método sencillo y fácil de entender, pero su eficiencia disminuye linealmente a medida que el tamaño de la lista aumenta.

```
def busqueda_lineal_por_id(lista, id_buscar):
    for producto in lista:
        if producto.get('id_producto') == id_buscar:
            return producto
```

```
return None
```

Búsqueda Binaria (busqueda_binaria_por_id): Esta función implementa el algoritmo de búsqueda binaria. A diferencia de la lineal, esta requiere que la lista de productos esté previamente ordenada por la clave de búsqueda (id_producto en este caso). Funciona dividiendo repetidamente por la mitad la porción de la lista donde el elemento podría estar, eliminando la mitad que no puede contener el valor. Este proceso continúa hasta que el elemento es encontrado o se determina que no existe. Su eficiencia es significativamente mayor que la lineal para listas grandes, como se demuestra en las pruebas de tiempo.

```
def busqueda_binaria_por_id(lista_ordenada, id_buscar):
    izquierda, derecha = 0, len(lista_ordenada) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        producto_medio = lista_ordenada[medio]
         id_medio = producto_medio.get('id_producto')
        if id_medio == id_buscar:
            return producto_medio
        elif id_medio < id_buscar:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return None
```

El corazón de la demostración reside en la implementación de tres algoritmos de ordenamiento clave, todos diseñados para ordenar la lista de productos basándose en la clave precio (por defecto, en orden ascendente):

Bubble Sort (bubble_sort): Este algoritmo es uno de los más simples de entender. Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite en pasadas sucesivas hasta que la lista está completamente ordenada. Se caracteriza por su baja eficiencia para grandes conjuntos de datos, ya que realiza muchas comparaciones e intercambios.

```
def bubble_sort(lista, clave='precio', reverse=False):
    n = len(lista)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            val1 = lista[j].get(clave)
            val2 = lista[j+1].get(clave)
            if reverse: # Orden descendente
                if val1 < val2:
                    lista[j], lista[j+1] = lista[j+1], lista[j]
            else: # Orden ascendente
                if val1 > val2:
                    lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
```

Insertion Sort (insertion_sort): Este algoritmo construye la lista ordenada un elemento a la vez. Toma elementos de la porción no ordenada de la lista y los inserta en su posición correcta dentro de la porción ya ordenada. Es eficiente para listas pequeñas o listas que ya están casi ordenadas, pero su rendimiento disminuye con listas grandes de forma similar al Bubble Sort.

```
def insertion_sort(lista, clave='precio', reverse=False):
    for i in range(1, len(lista)):
        actual = lista[i]
        j = i - 1
        while j >= 0:
            if reverse: # Orden descendente
                if actual.get(clave) > lista[j].get(clave):
                    lista[j+1] = lista[j]
                    j -= 1
                else:
                    break
            else: # Orden ascendente
                if actual.get(clave) < lista[j].get(clave):
```

```
            lista[j+1] = lista[j]
            j -= 1
        else:
            break
    lista[j+1] = actual
return lista
```

Merge Sort (merge_sort y _merge): Este es un algoritmo de ordenamiento más avanzado y eficiente, basado en el paradigma "Divide y Vencerás". Funciona dividiendo recursivamente la lista en dos mitades hasta que cada sublista contiene un solo elemento (que por definición está ordenado). Luego, combina (_merge) estas sublistas de forma ordenada. Es conocido por su eficiencia consistente y predecible, lo que lo hace adecuado para grandes volúmenes de datos.

```
def merge_sort(lista, clave='precio', reverse=False):
    if len(lista) <= 1:
        return lista
    medio = len(lista) // 2
    izquierda = lista[:medio]
    derecha = lista[medio:]
    izquierda = merge_sort(izquierda, clave, reverse)
    derecha = merge_sort(derecha, clave, reverse)
    return _merge(izquierda, derecha, clave, reverse)
def _merge(izquierda, derecha, clave, reverse):
    resultado = []
    i = j = 0

    while i < len(izquierda) and j < len(derecha):
        val_izq = izquierda[i].get(clave)
        val_der = derecha[j].get(clave)

        if reverse: # Descendente
```

```
            if val_izq >= val_der:
                resultado.append(izquierda[i])
                i += 1
            else:
                resultado.append(derecha[j])
                j += 1
        else: # Ascendente
            if val_izq <= val_der:
                resultado.append(izquierda[i])
                i += 1
            else:
                resultado.append(derecha[j])
                j += 1


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

Es importante destacar que, para asegurar una medición justa de los tiempos de cada algoritmo de ordenamiento, cada función de ordenamiento recibe una copia profunda (copy.deepcopy()) del inventario original. Esto garantiza que cada algoritmo inicie su proceso con un conjunto de datos desordenado idéntico, evitando que un ordenamiento previo afecte las mediciones de los algoritmos subsiguientes.

La función ejecutar_pruebas() orquesta toda la demostración y es el punto de entrada principal del script. Sus pasos clave son:

Generación de Inventarios Base: Primero, se generan tres conjuntos de datos de inventario (inventario_base_pequeno, inventario_base_mediano, inventario_base_grande) con diferentes cantidades de productos. Estos inventarios originales y desordenados servirán como base para todas las pruebas.

Pruebas de Ordenamiento: Para cada tamaño de inventario base, el script itera a través de los algoritmos de ordenamiento implementados (Bubble Sort, Insertion Sort, Merge Sort y el Timsort incorporado de Python). Antes de ejecutar cada algoritmo, se crea una copia del inventario base original para asegurar que el algoritmo siempre opere sobre datos desordenados. Se mide el tiempo de ejecución de cada ordenamiento utilizando time.time() y los resultados se imprimen en consola, permitiendo una comparación directa de la eficiencia. Para evitar tiempos de espera excesivos, los algoritmos de baja eficiencia (Bubble Sort e Insertion Sort) se omiten en los inventarios de mayor tamaño.

```
ordenamientos = {
    "Bubble Sort": bubble_sort,
    "Insertion Sort": insertion_sort,
    "Merge Sort": merge_sort,
        "Python Timsort (built-in)": lambda lst, clave='precio', reverse=False: sorted(lst, key=lambda x: x.get(clave), reverse=reverse)
}
inventarios_a_probar = {
    "Inventario Pequeño (100)": inventario_base_pequeno,
    "Inventario Mediano (4500)": inventario_base_mediano,
    "Inventario Grande (15000)": inventario_base_grande
}

for tamano_desc, inv_original in inventarios_a_probar.items():
    print(f"\n--- Probando con {tamano_desc} ---")
    for nombre_algo, algo_func in ordenamientos.items():
        inventario_para_prueba = copy.deepcopy(inv_original)

        start_time = time.time()
        _ = algo_func(inventario_para_prueba, clave='precio', reverse=False)
        end_time = time.time()
        print(f"  {nombre_algo}: {end_time - start_time:.6f} segundos")
```

Pruebas de Búsqueda: Utilizando el inventario más grande para una mejor apreciación de las diferencias, se realizan pruebas de búsqueda lineal y binaria por id_producto. Para la búsqueda binaria, se crea una versión del inventario previamente ordenada por ID utilizando sorted() de Python (que internamente usa Timsort), ya que la búsqueda binaria lo requiere. Se miden y comparan los tiempos para buscar tanto un ID existente como uno no existente, ilustrando la ventaja de la búsqueda binaria en listas ordenadas.

```
print("\n--- Búsqueda Lineal por ID ---")

start_time = time.time()

resultado_lineal_existente = busqueda_lineal_por_id(inventario_busqueda, id_existente)

end_time = time.time()

print(f" Búsqueda Lineal por ID '{id_existente}' (existente): {end_time - start_time:.9f} segundos. Encontrado: {'Sí' if resultado_lineal_existente else 'No'}")


start_time = time.time()

resultado_lineal_no_existente = busqueda_lineal_por_id(inventario_busqueda, id_no_existente)

end_time = time.time()

print(f" Búsqueda Lineal por ID '{id_no_existente}' (no existente): {end_time - start_time:.9f} segundos. Encontrado: {'Sí' if resultado_lineal_no_existente else 'No'}")


print("\n--- Búsqueda Binaria por ID (en lista ordenada) ---")

start_time = time.time()

resultado_binaria_existente= busqueda_binaria_por_id(inventario_ordenado_por_id, id_existente)

end_time = time.time()

print(f" Búsqueda Binaria por ID '{id_existente}' (existente): {end_time - start_time:.9f} segundos. Encontrado: {'Sí' if resultado_binaria_existente else 'No'}")


start_time = time.time()

resultado_binaria_no_existente = busqueda_binaria_por_id(inventario_ordenado_por_id, id_no_existente)

end_time = time.time()
```

```
print(f"  Búsqueda Binaria por ID '{id_no_existente}' (no existente): {end_time - start_time:.9f} segundos. Encontrado: {'Sí' if resultado_binaria_no_existente else 'No'}")
```

En resumen, este código proporciona una plataforma práctica para observar y analizar el rendimiento de algoritmos fundamentales, destacando la importancia de la elección del algoritmo adecuado en función del volumen de datos y los requisitos de eficiencia operativa.

Metodología Utilizada

El presente trabajo se llevó a cabo siguiendo un enfoque estructurado que combinó investigación teórica con aplicación práctica en Python. En primer lugar, se realizó una búsqueda exhaustiva de información bibliográfica en fuentes confiables, como libros de algoritmos y documentación oficial de Python. Estas fuentes permitieron comprender en profundidad los fundamentos de los algoritmos de búsqueda y ordenamiento, así como sus aplicaciones y complejidades.

Una vez consolidada la base teórica, se inició la etapa de diseño y desarrollo del código. Esta fase se dividió en la implementación individual de cada algoritmo (Bubble Sort, Insertion Sort, Merge Sort, búsqueda lineal y binaria), su explicación detallada en párrafos, y la validación de su funcionamiento con listas reales. Posteriormente, se desarrollaron casos prácticos específicos (comparación por tamaño de lista, listas ordenadas e inversas, estructuras con objetos y Timsort educativo) para comparar empíricamente el rendimiento de los algoritmos. Para ello, se utilizaron mediciones de tiempo de ejecución y análisis de comportamiento en distintos escenarios.

Las herramientas utilizadas durante el desarrollo fueron el lenguaje Python 3, junto con las bibliotecas random para generar datos de prueba y time para medir el rendimiento de los algoritmos. Este enfoque integral permitió no solo una comprensión conceptual, sino también una evaluación práctica y visual de los algoritmos estudiados.

Resultados Obtenidos

Esta sección presenta los resultados de tiempo de ejecución de los algoritmos de ordenamiento y búsqueda implementados, ofreciendo una comparación directa de su eficiencia. Las pruebas se realizaron con inventarios de diferentes tamaños para ilustrar cómo el rendimiento de cada algoritmo escala con la cantidad de datos. Los tiempos se expresan en segundos, mostrando la fracción de tiempo que tardó cada operación.

Pruebas de Ordenamiento

En esta sección, probamos el rendimiento de Bubble Sort, Insertion Sort, Merge Sort, y el algoritmo Timsort (usado internamente por Python con sorted()) al ordenar el inventario por el campo precio en orden ascendente.

Inventario Pequeño (100 Productos):

- Bubble Sort: 0.000000 segundos
- Insertion Sort: 0.000309 segundos
- Merge Sort: 0.000000 segundos
- Python Timsort: 0.000000 segundos

Para un conjunto de datos reducido como 100 productos, las diferencias de tiempo entre los algoritmos son mínimas, a menudo en el rango de microsegundos. La precisión de time.time() puede incluso redondear a cero para operaciones extremadamente rápidas. Esto demuestra que para listas muy pequeñas, la elección del algoritmo de ordenamiento tiene un impacto despreciable en el rendimiento. Sin embargo, ya se puede intuir la eficiencia superior de Merge Sort y Timsort, que son intrínsecamente más optimizados.

Inventario Mediano (4500 Productos):

- Bubble Sort: 4.625716 segundos
- Insertion Sort: 2.058161 segundos
- Merge Sort: 0.036046 segundos

- Python Timsort: 0.002930 segundos

Aquí es donde las diferencias en la eficiencia algorítmica se vuelven dramáticamente evidentes. Bubble Sort e Insertion Sort, con una complejidad temporal de O(n2) (cuadrática), muestran un incremento sustancial en el tiempo de ejecución, tardando varios segundos. Su rendimiento decae rápidamente a medida que el número de elementos aumenta. Por otro lado, Merge Sort, con una complejidad de O(nlogn) (logarítmica lineal), completa la tarea en una fracción de segundo. El Timsort de Python, siendo un algoritmo híbrido altamente optimizado, es aún más rápido, demostrando una eficiencia excepcional para este tamaño de datos.

Inventario Grande (15000 Productos):

- Bubble Sort: 49.198748 segundos
- Insertion Sort: 24.072007 segundos
- Merge Sort: 0.147990 segundos
- Python Timsort (built-in): 0.011000 segundos

Para un inventario de 15,000 productos, la diferencia de rendimiento es abrumadora. Bubble Sort e Insertion Sort tardan decenas de segundos, volviéndose completamente imprácticos para aplicaciones reales. Esto subraya la naturaleza de su complejidad cuadrática, donde un pequeño aumento en n resulta en un gran aumento en el tiempo. En contraste, Merge Sort maneja el volumen de datos con gran solvencia, completando la ordenación en décimas de segundo. Timsort mantiene su liderazgo como el más eficiente, demostrando ser la opción más robusta y escalable para grandes conjuntos de datos.

Pruebas de Búsqueda

En esta sección, comparamos la búsqueda lineal con la búsqueda binaria al intentar localizar un producto específico por su id_producto en un inventario grande (15,000 productos). Es importante recordar que la búsqueda binaria solo puede realizarse en una lista que ya está ordenada por la clave de búsqueda.

Inventario Grande (15000 Productos):

Búsqueda Lineal:

- Buscar ID 'PROD07500' (existente): 0.000915766 segundos.
    - *Resultado:* Encontrado: Sí
- Buscar ID 'NO_EXISTE_12345' (no existente): 0.002070427 segundos.
    - *Resultado:* Encontrado: No

La búsqueda lineal recorre la lista elemento por elemento. Como se observa, el tiempo para encontrar un elemento existente puede ser menor que para uno no existente o que se encuentra al final de la lista, ya que la búsqueda termina tan pronto como se encuentra el elemento. Sus tiempos, aunque pequeños para una sola búsqueda, son directamente proporcionales al tamaño de la lista O(n) en el peor caso.

Búsqueda Binaria (en lista ordenada por ID):

- Buscar ID 'PROD07500' (existente): 0.000000000 segundos.
    - *Resultado:* Encontrado: Sí
- Buscar ID 'NO_EXISTE_12345' (no existente): 0.000000000 segundos.
    - *Resultado:* Encontrado: No

La búsqueda binaria, al operar sobre una lista previamente ordenada, es significativamente más rápida que la lineal para grandes volúmenes de datos. Su complejidad temporal es O(logn), lo que significa que el tiempo de búsqueda aumenta muy poco a medida que la lista crece. Como se observa en los resultados, los tiempos de búsqueda son extremadamente bajos (a menudo redondeados a cero por la alta precisión del cronómetro), tanto para IDs existentes como para los no existentes. Esto demuestra la vasta eficiencia que se obtiene al aprovechar una estructura de datos ordenada para la búsqueda.

Conclusiones

Este trabajo permitió comparar en la práctica diferentes algoritmos de búsqueda y ordenamiento. La eficiencia de un algoritmo depende en gran parte del contexto: volumen de datos, necesidad de orden previo, y complejidad computacional. Los datos recolectados ilustran de manera contundente la

importancia de la elección del algoritmo en función de la escala de los datos. Para la ordenación, algoritmos con complejidad O(nlogn) como Merge Sort y Timsort son cruciales para manejar grandes volúmenes de información de manera eficiente, mientras que los algoritmos de complejidad O(n2) como Bubble Sort e Insertion Sort se vuelven rápidamente inviables. En cuanto a la búsqueda, la búsqueda binaria O(logn) demuestra una eficiencia muy superior a la búsqueda lineal O(n) en conjuntos de datos grandes, siempre que los datos estén previamente ordenados. Esto subraya cómo una inversión inicial en ordenar los datos puede traducirse en una mejora drástica en la eficiencia de las operaciones de búsqueda posteriores, optimizando así el rendimiento general de un sistema de gestión de inventario.

Bibliografía

Python Software Foundation. (2024). Python 3 Documentation.

https://docs.python.org/3/

Sánchez Ubeda, E. F. (2001). Algoritmos de ordenación (II).

CORTEZ GALVAN, L. E. (2024). ALGORITMOS DE ORDENAMIENTO.

Anexos

Link GitHub del repositorio:

https://github.com/Diexvel/IntegradorProgramacion.git

Link Video:

https://drive.google.com/file/d/1HvroqZ7VJ_c7IeWgIAcz9ktrCgbk_cGc/view?usp=drive_link

Captura de Salida de consola: