Universidad Americana

Asignatura: Algoritmos y Estructura de Datos

Proyecto de Investigación Final

Autores:

Julio César Méndez Sandigo

Diego Alejandro Palacios

Iván Castillo

Docente:

Silvia Gigdalia Ticay López

30 de junio de 2025

Índice de Contenido

I. Introducción

La eficiencia computacional es fundamental en el desarrollo de softwares y sistemas informáticos. En un mundo cada vez más dependiente de la tecnología, la capacidad de procesar grandes volúmenes de datos de manera rápida y eficiente se ha vuelto crucial. Los algoritmos de ordenamiento y búsqueda son esenciales en este contexto, ya que permiten organizar y localizar información dentro de estructuras de datos de manera óptima. La elección del algoritmo adecuado puede tener un impacto significativo en el rendimiento de una aplicación, afectando directamente el tiempo de ejecución y el consumo de recursos.

El presente trabajo de investigación se centra en el análisis de dos algoritmos básicos: el algoritmo de ordenamiento Bubble Sort y el algoritmo de búsqueda Búsqueda Secuencial en arreglos desordenados. A través de esta investigación, se busca comprender su funcionamiento, características y eficiencia computacional, tanto en términos de tiempo como de espacio. Se expondrám conceptos clave como el análisis a priori y a posteriori, la notación Big O, y se presentarán ejemplos de sus implementaciones conceptuales.

Aunque existen algoritmos más avanzados y eficientes para realizar tareas de ordenamiento y búsqueda, el estudio de Bubble Sort y la Búsqueda Secuencial resulta invaluable para establecer una base sólida en el entendimiento de la complejidad algorítmica. Estos algoritmos, a pesar de su simplicidad, ofrecen una excelente oportunidad para ilustrar los principios de diseño y análisis de algoritmos, así como para reflexionar sobre las implicaciones que tiene la elección de un algoritmo en el rendimiento de un sistema.

Planteamiento del problema

En el ámbito de la informática y la ciencia de datos, la gestión eficiente de la información es un desafío constante. Las aplicaciones modernas, desde bases de datos hasta sistemas de inteligencia artificial, dependen en gran medida de la capacidad de ordenar y buscar datos de manera efectiva. Sin embargo, la elección de algoritmos de ordenamiento y búsqueda no es trivial, ya que su desempeño puede variar drásticamente según el tamaño y la naturaleza de los datos, así como las condiciones de ejecución. Un algoritmo ineficiente puede llevar a tiempos de respuesta inaceptablemente largos, un consumo excesivo de recursos computacionales y, en última instancia, a una experiencia de usuario deficiente o a la inviabilidad de la aplicación.

La necesidad de evaluar la eficiencia de los algoritmos de ordenamiento y búsqueda surge de la observación de que su rendimiento impacta directamente en el tiempo de ejecución y el uso de recursos en aplicaciones reales. Por ejemplo, en un sistema de gestión de inventario con millones de productos, una búsqueda ineficiente podría paralizar las operaciones, mientras que un ordenamiento lento podría retrasar la generación de informes críticos. Esta problemática se agrava con el crecimiento exponencial de los datos, lo que exige soluciones algorítmicas que escalan de manera adecuada.

Por lo tanto, es fundamental comprender las características de desempeño de los algoritmos más comunes, como Bubble Sort y la Búsqueda Secuencial, para poder tomar decisiones informadas sobre su aplicación. Aunque estos algoritmos son conceptualmente simples, su análisis detallado revela las limitaciones y ventajas que los hacen más o menos adecuados para diferentes escenarios. La falta de un conocimiento profundo sobre su eficiencia puede conducir a la implementación de soluciones subóptimas, lo que subraya la importancia de esta investigación para proporcionar una base sólida para la selección y optimización de algoritmos en el desarrollo de software.

Objetivo de la investigación

Objetivo general

Analizar la eficiencia computacional de los algoritmos de ordenamiento "Bubble Sort" y de búsqueda "Búsqueda Secuencial en arreglos desordenados" mediante la exploración de su funcionamiento, características, y el estudio de su rendimiento en términos de tiempo y espacio.

Objetivos específicos

Describir en detalle el funcionamiento interno y las características fundamentales del algoritmo Bubble Sort.

Explicar el proceso de la Búsqueda Secuencial en arreglos desordenados, destacando sus particularidades.

Evaluar la eficiencia espacial y temporal de ambos algoritmos utilizando la notación Big O, considerando sus mejores, promedios y peores casos.

Presentar códigos claros y concisos para la implementación conceptual de Bubble Sort y la Búsqueda Secuencial.

Comparar los resultados obtenidos en función de la complejidad de ambos algoritmos en dos casos de aplicación.

II. Metodología

La metodología de esta investigación se diseñó para analizar y comparar la eficiencia computacional de los algoritmos Bubble Sort y Búsqueda Secuencial en arreglos desordenados. Se emplea un enfoque experimental y cuantitativo para recolectar y analizar datos de rendimiento.

A. Tipo y Diseño de Investigación

Esta investigación es de tipo experimental, ya que implica la manipulación controlada de variables, específicamente estructuras de datos y algoritmos, para medir su impacto en el rendimiento. Se realizaron pruebas controladas para evaluar la eficiencia de los algoritmos en diferentes escenarios.

B. Enfoque de la Investigación

El enfoque de la investigación es cuantitativo, lo que significa que se recolectarán y analizarán datos numéricos y medibles. Esto incluye métricas como el tiempo de ejecución y el uso de memoria, que permitirán una comparación objetiva del desempeño de los algoritmos.

C. Alcance de la Investigación

El alcance de este estudio es descriptivo y explicativo. Es descriptivo porque se detallarán las características y el funcionamiento de los algoritmos Bubble Sort y Búsqueda Secuencial. Es explicativo porque se buscará comprender y justificar el comportamiento de estos algoritmos frente a diferentes tipos y tamaños de entradas de datos.

D. Procedimiento

El procedimiento para llevar a cabo esta investigación se estructurará en las siguientes fases:

1. **Selección de Algoritmos**: Se seleccionarán específicamente el algoritmo de ordenamiento Bubble Sort y el algoritmo de búsqueda Secuencial en arreglos desordenados para su estudio.

2. **Codificación e Implementación**: Ambos algoritmos serán codificados e implementados en un lenguaje de programación de alto nivel (por ejemplo, Python). Se documentará el código fuente y se incluirán explicaciones de su funcionamiento y estructura, pudiendo presentarse fragmentos de código, diagramas de flujo o pseudocódigo.

3. **Generación de Datos de Prueba**: Se crearán conjuntos de datos con diferentes tamaños (ej., pequeños, medianos, grandes) y características (ej., datos ya ordenados, datos aleatorios, datos en orden inverso) para evaluar los algoritmos en diversos escenarios.

4. **Ejecución y Medición de Eficiencia**: Los algoritmos implementados se ejecutarán utilizando los distintos tamaños y tipos de datos de prueba. Se medirán métricas de rendimiento clave como el tiempo de ejecución (eficiencia temporal) y el consumo de memoria (eficiencia espacial).

5. **Análisis a Priori**: Antes de la ejecución, se realizará un análisis teórico de la eficiencia espacial y temporal de cada algoritmo, estimando la cantidad de memoria y el número de operaciones necesarias, respectivamente. Esto incluirá la clasificación de los algoritmos en notación Big O ($O(n)$, $O(\log n)$, $O(n^2)$, etc.).

6. **Análisis a Posteriori**: Después de la ejecución, se analizarán los datos recolectados para el mejor caso (condiciones ideales, como una lista ya ordenada para Bubble Sort), el caso promedio (rendimiento típico con entradas aleatorias), y el peor caso (situación menos favorable, como una lista en orden inverso).

7. **Análisis Comparativo**: Se comparan los resultados obtenidos para ambos algoritmos, prestando atención a las diferencias en su eficiencia en distintos escenarios y el impacto de la complejidad algorítmica. Los resultados se presentarán en tablas y gráficas para facilitar su visualización y análisis cuantitativo.

III. Marco conceptual / referencial

Marco Teórico sobre Eficiencia Computacional

1. Algoritmo

1.1. Definición formal

Un algoritmo es un conjunto finito de instrucciones precisas y ordenadas que permiten resolver un problema específico mediante una secuencia de pasos que transforman una entrada en una salida. Esta secuencia debe estar claramente definida, de modo que no genere ambigüedades, y debe ser finita, lo que implica que eventualmente debe concluir. Cada algoritmo está diseñado para abordar un tipo de problema determinado, y puede tener diferentes niveles de eficiencia dependiendo de su estructura lógica y del tratamiento de datos. Como explican Olawanle (2024) y Canquil (s.f.), los algoritmos constituyen la base del pensamiento computacional y permiten automatizar tareas, siendo la notación Big-O una herramienta crucial para analizar su eficiencia.

1.2. Propiedades esenciales

Todo algoritmo debe cumplir con ciertas propiedades fundamentales para ser considerado válido: finitud, precisión, entrada, salida y secuencia definida. La finitud garantiza que el algoritmo termina en un número limitado de pasos. La precisión implica que cada instrucción está definida de manera inequívoca. Las entradas representan los datos iniciales sobre los cuales se realiza el procesamiento, mientras que las salidas son los resultados obtenidos tras la ejecución del algoritmo. Finalmente, la secuencia definida exige que las instrucciones se ejecuten en un orden lógico preestablecido. Estas propiedades aseguran que el algoritmo sea ejecutable y que produzca resultados confiables y repetibles.

2. Orden en los algoritmos

2.1. Relación de orden

En los algoritmos que manipulan estructuras de datos, es común utilizar relaciones de orden para comparar elementos. Estas relaciones pueden ser numéricas,

cuando los datos son valores enteros o reales, o lexicográficas, cuando se trata de cadenas de texto. Establecer una relación de orden adecuada es fundamental para algoritmos de ordenamiento y búsqueda, ya que permite determinar si un elemento es mayor, menor o igual a otro según ciertos criterios. Aunque no siempre se menciona de forma explícita en la literatura, esta relación está implícita en la lógica de comparación de muchos algoritmos.

2.2. Orden de crecimiento

El orden de crecimiento se refiere a la manera en que aumentan los recursos necesarios (tiempo y espacio) para ejecutar un algoritmo conforme incrementa el tamaño de la entrada. Este concepto es evaluado mediante notaciones asintóticas como O, Ω y Θ, que permiten categorizar los algoritmos según su eficiencia. Como explica DataCamp (s.f.), las clases más comunes de complejidad incluyen:

- O(1): tiempo constante, independiente del tamaño de la entrada.
- O(log n): tiempo logarítmico, eficiente para entradas grandes.
- O(n): tiempo lineal, razonable en la mayoría de los casos.
- O(n log n): tiempo logarímico-lineal, común en algoritmos de ordenamiento eficientes.
- O(n^2), O(2^n), O(n!): tiempos cuadrático, exponencial y factorial, que resultan ineficientes para datos masivos.

Estas categorías permiten a los desarrolladores anticipar el comportamiento de los algoritmos sin necesidad de pruebas empíricas.

3. Complejidad temporal y espacial

3.1. Complejidad temporal

La complejidad temporal mide la cantidad de tiempo que tarda un algoritmo en completarse en función del tamaño de la entrada. Es una métrica esencial para analizar el rendimiento de una solución algorítmica. Olawanle (2024) subraya que esta medida

permite estimar la eficiencia de un algoritmo sin necesidad de implementarlo. La complejidad temporal ayuda a identificar cuántas operaciones se necesitan en el mejor, peor o promedio de los casos, lo que facilita una elección informada entre varias alternativas de resolución.

3.2. Complejidad espacial

La complejidad espacial, por otro lado, se refiere a la cantidad de memoria adicional que un algoritmo requiere para ejecutarse. Esto incluye variables, estructuras auxiliares y el espacio necesario para llamadas recursivas. Guano (s.f.) enfatiza que en entornos con recursos limitados, la eficiencia espacial se vuelve crucial. Algoritmos que consumen mucha memoria pueden ser inviables en dispositivos con poca capacidad, a pesar de tener buena eficiencia temporal.

3.3. Notaciones asintóticas

Las notaciones asintóticas son expresiones matemáticas que describen el comportamiento de los algoritmos a medida que el tamaño de entrada crece hacia el infinito. La notación Big-O representa la cota superior del tiempo de ejecución, es decir, el peor escenario posible. La notación Ω (Omega) define la cota inferior, o el mejor de los casos. Finalmente, Θ (Theta) describe la cota ajustada, es decir, cuando el mejor y peor caso tienen el mismo orden de crecimiento. Estas notaciones permiten abstraer el comportamiento del algoritmo y enfocarse en su eficiencia relativa.

4. Análisis a priori

4.1. Definición

El análisis a priori es una técnica teórica que permite estimar la eficiencia de un algoritmo antes de ser implementado. Según Dialgo (2012), este método se basa en una descripción formal del algoritmo y en el conteo de operaciones, permitiendo predecir su rendimiento sin realizar pruebas. Es una herramienta fundamental para elegir entre diferentes estrategias algorítmicas durante la fase de diseño de software.

4.2. Objetivos y ventajas

El principal objetivo del análisis a priori es anticipar posibles problemas de eficiencia y seleccionar la opción más conveniente. Entre sus ventajas se encuentra la posibilidad de comparar algoritmos de manera objetiva, evitando costosos procesos de prueba y error. Además, este tipo de análisis facilita la documentación y justificación de las decisiones técnicas tomadas durante el desarrollo.

5. Análisis a posteriori

5.1. Definición

El análisis a posteriori, a diferencia del anterior, consiste en medir el comportamiento real del algoritmo una vez implementado. Como explica Sunter (2022), este tipo de análisis se realiza ejecutando el algoritmo en un entorno de prueba, registrando tiempos de ejecución y uso de recursos para distintos tamaños de entrada.

5.2. Ventajas y limitaciones

Entre las ventajas del análisis a posteriori se encuentra su precisión en condiciones reales. Sin embargo, sus resultados pueden variar según el entorno de ejecución (hardware, sistema operativo, lenguaje de programación), lo cual limita su generalización. A pesar de ello, es una herramienta valiosa para validar las estimaciones teóricas obtenidas mediante el análisis a priori.

6. Casos de análisis

6.1. Mejor caso, caso promedio, peor caso

El comportamiento de un algoritmo puede cambiar significativamente dependiendo del tipo de datos que reciba como entrada. Dialgo (2012) distingue entre el mejor caso, en el que se realizan la menor cantidad posible de operaciones; el peor caso, en el que se ejecuta el máximo número de pasos; y el caso promedio, que representa un valor esperado considerando todas las entradas posibles. Analizar estos tres escenarios permite tener una visión completa del rendimiento del algoritmo.

6.2. Importancia en la evaluación

La evaluación basada en estos casos es esencial para seleccionar el algoritmo más adecuado en función del contexto de uso. En sistemas críticos, por ejemplo, es común diseñar para el peor caso, garantizando que el rendimiento no caerá por debajo de cierto umbral.

7. Comparativa de algoritmos

7.1. Criterios de comparación

Al comparar algoritmos, se toman en cuenta diversos criterios como el tiempo de ejecución, uso de memoria, estabilidad (en algoritmos de ordenamiento), adaptatividad (capacidad para ajustarse a la estructura de los datos) y facilidad de implementación. Olawanle (2024) destaca que los factores de tiempo y espacio suelen ser los más relevantes, aunque otros aspectos como la legibilidad del código también pueden influir en contextos de desarrollo colaborativo.

7.2. Métodos de comparación

Dialgo (2012) sugiere que la comparación ideal se basa en una combinación de análisis a priori y a posteriori. Mientras el primero permite anticipar el rendimiento teórico, el segundo confirma estos resultados en condiciones reales. Esta doble perspectiva facilita una selección más acertada del algoritmo adecuado para un problema particular.

8. Aplicación específica: Bubble Sort y Búsqueda Secuencial

8.1. Descripción del funcionamiento

Bubble Sort: Es un algoritmo de ordenamiento que funciona comparando pares de elementos adyacentes e intercambiándose si están en el orden incorrecto. Este proceso se repite hasta que no hay más cambios, lo que indica que la lista está ordenada. Según Campos (2024), es un algoritmo fácil de entender e implementar, lo que lo convierte en una herramienta educativa muy utilizada.

Búsqueda secuencial: por su parte, es un algoritmo de búsqueda que revisa uno a uno los elementos de una lista hasta encontrar el valor deseado o hasta terminar el recorrido. CertiDevs (2025) lo describe como un método simple, ideal para listas pequeñas o sin orden preestablecido.

8.2. Complejidad temporal y espacial

El algoritmo Bubble Sort tiene una complejidad temporal de O(n^2) en el peor y promedio de los casos, mientras que su complejidad espacial es O(1), ya que no requiere memoria adicional significativa (Campos, 2024). Por otro lado, la búsqueda secuencial tiene una complejidad temporal de O(n) en el peor caso y de O(1) en el mejor caso, con una complejidad espacial también de O(1) (CertiDevs, 2025).

8.3. Enfoques a priori vs. a posteriori aplicados

El análisis a priori permite estimar las complejidades mencionadas sin necesidad de pruebas. Sin embargo, como afirma Sunter (2022), el análisis a posteriori es clave para validar estas predicciones mediante pruebas reales, especialmente cuando se buscan optimizaciones específicas en implementaciones concretas.

8.4. Comparativa entre ambos

FasterCapital (2025) concluye que Bubble Sort, a pesar de su ineficiencia en grandes volúmenes de datos, es útil por su simplicidad y facilidad de implementación. En contraste, la búsqueda secuencial es más eficiente para listas desordenadas de tamaño moderado y no requiere estructuras auxiliares ni ordenamiento previo. La elección entre ambos dependerá del contexto específico y del tipo de problema que se desea resolver.

IV. Implementación del algoritmo

La implementación de los algoritmos estudiados en el presente proyecto se llevó a cabo en el lenguaje de programación Python, distribuyendo la lógica en módulos separados para garantizar la claridad y modularidad del código. En particular, se implementaron los algoritmos de ordenamiento Bubble Sort y de Búsqueda Secuencial, ambos aplicados sobre estructuras de datos definidas en clases personalizadas. A continuación, se describen las secciones más relevantes de dicha implementación.

Implementación del algoritmo Bubble Sort

La implementación del algoritmo Bubble Sort se ha realizado en el lenguaje de programación Python, haciendo uso de estructuras de datos diseñadas específicamente para este proyecto, y distribuidas en módulos independientes que permiten una mejor organización del código. El enfoque utilizado se basa en listas doblemente enlazadas, lo cual representa una alternativa más dinámica a los arreglos estáticos tradicionales, permitiendo operar sobre nodos con múltiples atributos sin necesidad de reestructurar la memoria.

Módulo clases.py: definición de nodos y lista enlazada

El archivo clases.py constituye el pilar estructural de la implementación, ya que define tanto la clase Nodo —que representa cada tarea individual— como la clase ListaDobleEnlazada, que permite gestionar toda la colección de tareas. Este diseño basado en programación orientada a objetos facilita la manipulación ordenada de datos y su posterior ordenamiento con Bubble Sort.

Cada nodo contiene:

- titulo: texto que identifica la tarea.
- prioridad: número entero que indica su nivel de urgencia.
- fecha_limite: fecha límite convertida al tipo datetime para facilitar comparaciones.
- anterior y siguiente: punteros dobles que enlazan el nodo con sus vecinos inmediatos.

La clase ListaDobleEnlazada incluye métodos fundamentales como agregar_tarea() (para insertar nodos al final), mostrar_tareas() (para visualizar la lista), buscar_tarea() (para localizar una tarea por su título) y modificar_tarea() (para editar los valores de una tarea existente). Estos métodos garantizan una gestión completa de las tareas previo al ordenamiento.

```python
class Nodo:
    def __init__(self, titulo, prioridad, fecha_limite):
        self.titulo = titulo
        self.prioridad = prioridad
        self.fecha_limite = datetime.strptime(fecha_limite, "%Y-%m-%d")
        self.anterior = None
        self.siguiente = None
```

```python
class ListaDobleEnlazada:
    def __init__(self):
        self.cabeza = None

    def agregar_tarea(self, titulo, prioridad, fecha_limite):···

    def mostrar_tareas(self):···

    def buscar_tarea(self, titulo_busqueda):···

    def modificar_tarea(self, titulo_busqueda, nuevo_titulo, nueva_prioridad, nueva_fecha):··
```

Módulo bubble_sort.py: lógica del algoritmo

El archivo bubble_sort.py contiene la función bubble_sort(lista), la cual constituye el núcleo del algoritmo de ordenamiento. Esta función opera sobre una lista doblemente enlazada y aplica el método de burbuja, que se basa en comparar pares de nodos adyacentes y reorganizar su contenido si están en un orden incorrecto.

La lógica implementada sigue una estructura iterativa que recorre toda la lista repetidamente mientras se detecten cambios. El criterio de comparación es jerárquico: primero se ordena por prioridad descendente (es decir, tareas más importantes al inicio), y en caso de empate, se utiliza la fecha límite ascendente (las más próximas primero).

A diferencia de versiones tradicionales que intercambian posiciones en un arreglo, esta implementación intercambia solo los datos internos de los nodos, sin modificar sus referencias de enlace. Esto reduce el riesgo de errores y mantiene intacta la estructura de la lista.

```python
def bubble_sort(lista):
    if not lista.cabeza:
        return

    cambio = True
    while cambio:
        cambio = False
        actual = lista.cabeza
        while actual.siguiente:
            siguiente = actual.siguiente
            if (actual.prioridad < siguiente.prioridad) or \
               (actual.prioridad == siguiente.prioridad and actual.fecha_limite > siguiente.fecha_limite):
                # Intercambio de datos entre nodos
                actual.titulo, siguiente.titulo = siguiente.titulo, actual.titulo
                actual.prioridad, siguiente.prioridad = siguiente.prioridad, actual.prioridad
                actual.fecha_limite, siguiente.fecha_limite = siguiente.fecha_limite, actual.fecha_limite
                cambio = True
            actual = actual.siguiente
```

Módulo tareas_random.py: generación de datos de prueba

El archivo tareas_random.py contiene la función agregar_tareas_aleatorias(lista, cantidad), cuya finalidad es automatizar la creación de tareas con valores aleatorios. Esta función resulta fundamental para realizar pruebas empíricas del algoritmo en escenarios variados y con diferentes volúmenes de datos.

Cada tarea generada incluye un título dinámico, una prioridad aleatoria entre 1 y 5, y una fecha límite dentro de los próximos 30 días a partir del día actual. Estos parámetros simulan condiciones reales que pueden afectar el orden de ejecución de las tareas y permiten validar el funcionamiento del algoritmo con entradas no controladas.

```python
import random
from datetime import datetime, timedelta

def agregar_tareas_aleatorias(lista, cantidad):
    for i in range(cantidad):
        titulo = f"Tarea Aleatoria {i+1}"
        prioridad = random.randint(1, 5)  # Prioridad aleatoria entre 1 y 5
        dias_extra = random.randint(0, 30)  # Fechas dentro de los próximos 30 días
        fecha_limite = (datetime.today() + timedelta(days=dias_extra)).strftime("%Y-%m-%d")
        lista.agregar_tarea(titulo, prioridad, fecha_limite)
```

Módulo main.py: ejecución y validación del algoritmo

El archivo main.py representa el punto de integración de todos los módulos del proyecto. Su estructura define un menú interactivo mediante el cual el usuario puede realizar distintas acciones: agregar tareas manualmente, buscar tareas por título, modificar sus atributos, ver la lista completa, o aplicar el algoritmo de ordenamiento Bubble Sort.

De especial interés para esta implementación son las opciones 4 y 5 del menú:

Opción 4: permite visualizar el estado actual de la lista antes del ordenamiento.

Opción 5: ejecuta la función bubble_sort(lista) e informa al usuario que la lista ha sido ordenada exitosamente.

Además, la opción 7 permite generar tareas aleatorias que enriquecen las pruebas. Esta interacción directa entre el usuario y la lógica del algoritmo hace del módulo main.py una herramienta fundamental para validar el comportamiento del sistema y observar visualmente el efecto del algoritmo sobre los datos.

```
case "4":
    os.system('cls')
    tareas.mostrar_tareas()
    input("\nPresiona Enter para continuar...")

case "5":
    os.system('cls')
    print("Ordenando tareas por prioridad y fecha límite...")
    bubble_sort(tareas)
    print("Tareas ordenadas exitosamente.")
    input("\nPresiona Enter para continuar...")
```

Análisis final de la implementación del algoritmo Bubble Sort

La implementación del algoritmo Bubble Sort en este proyecto destaca por su correcta adaptación a una estructura de lista doblemente enlazada, lo cual representa una diferencia significativa respecto a las versiones más comunes que operan sobre arreglos. Este enfoque

estructural permitió trabajar con nodos que contienen múltiples atributos sin necesidad de manipular sus enlaces, lo cual contribuyó a mantener la estabilidad y coherencia del sistema.

La lógica de comparación jerárquica —prioridad seguida de fecha límite— permitió establecer un ordenamiento más realista, especialmente en contextos donde se gestiona una lista de tareas pendientes. Esta jerarquización introdujo mayor valor práctico al ordenamiento, permitiendo que las tareas más importantes y urgentes aparezcan primero.

Desde una perspectiva de diseño, el uso de módulos separados aumentó la legibilidad y mantenibilidad del código, facilitando futuras extensiones o modificaciones. Además, la incorporación de tareas aleatorias posibilitó pruebas más rigurosas, fundamentales para evaluar el algoritmo en condiciones diversas.

A pesar de que Bubble Sort no es el algoritmo más eficiente en términos de complejidad temporal ($O(n^2)$), su implementación en este contexto educativo permitió observar y analizar con claridad cómo afecta el orden de entrada al número de operaciones requeridas. Esta experiencia sienta una base sólida para comparaciones futuras con otros algoritmos de mayor eficiencia.

Implementación del Algoritmo en Busqueda Secuencial

La implementación del algoritmo de búsqueda secuencial se ha realizado en el lenguaje de programación Python, organizando la lógica en módulos separados para garantizar la claridad y la modularidad del código. En este proyecto, la búsqueda secuencial se aplica sobre una estructura de datos definida mediante una clase personalizada llamada PlaylistManager, la cual gestiona una lista de canciones almacenada en un archivo JSON.

El archivo playlist_manager.py constituye el pilar estructural de la implementación, ya que define la clase PlaylistManager encargada de toda la lógica de gestión de la colección de canciones. Este diseño orientado a objetos facilita operaciones como agregar, eliminar, mostrar y buscar canciones dentro de la playlist. Cada canción se representa como un diccionario con atributos clave: título, artista y álbum (opcional). La clase PlaylistManager incluye métodos fundamentales como añadir_cancion() para agregar nuevas canciones, mostrar_playlist() para mostrar la lista completa, eliminar_cancion() para eliminar una canción seleccionada por el

usuario y buscar_cancion() que implementa la búsqueda secuencial recorriendo toda la lista hasta encontrar una coincidencia por título.

```
class PlaylistManager:
    def __init__(self, archivo="playlist.json"):
        self.archivo = archivo
        self.playlist = []
        self.cargar_playlist()
```

El método buscar_cancion() solicita al usuario el nombre de la canción a buscar y recorre secuencialmente toda la lista. Para cada elemento, compara el título ingresado con el título almacenado, ignorando mayúsculas y minúsculas para mayor flexibilidad. Si encuentra una coincidencia, muestra la posición y la información de la canción; de lo contrario, informa que no existe en la playlist. Esta implementación destaca por su simplicidad y por no requerir estructuras de datos adicionales ni ordenamiento previo, manteniendo la lógica intuitiva y comprensible.

```
def buscar_cancion(self):
    objetivo = input("Nombre de la canción a buscar: ").strip().lower()
    for i, cancion in enumerate(self.playlist):
        if cancion['titulo'].lower() == objetivo:
            print(f'"{cancion["titulo"]}" de {cancion["artista"]} encontrada en la posición {i}.\n')
            return
    print(f'"{objetivo}" no está en la playlist.\n')
```

El archivo main.py integra todas las funcionalidades de PlaylistManager a través de un menú interactivo. Este menú permite al usuario gestionar la playlist mediante opciones como mostrar canciones, añadir nuevas, eliminar, reproducir y buscar. De especial relevancia para la búsqueda secuencial es la opción de buscar canción, que invoca el método buscar_cancion() para localizar rápidamente un título dentro de la lista. Esta interacción directa con el usuario hace del módulo main.py una herramienta clave para validar el funcionamiento del algoritmo y observar su rendimiento en listas de distintos tamaños.

```python
from playlist_manager import PlaylistManager, limpiar_pantalla

def mostrar_menu():
    print("1. Mostrar playlist")
    print("2. Añadir canción")
    print("3. Eliminar canción")
    print("4. Reproducir canción")
    print("5. Buscar canción")
    print("6. Salir")
```

La implementación de la búsqueda secuencial en este proyecto resalta por su adecuación a escenarios donde la lista de elementos no está ordenada y no se requiere un costo adicional de organización previa. Si bien su eficiencia en el peor caso es lineal (O(n)), su sencillez lo convierte en una solución práctica para listas pequeñas o medianas, como es el caso de una playlist personal. La modularidad del proyecto, la separación de responsabilidades en archivos independientes y la claridad del método buscar_cancion() facilitan su comprensión y posible mejora futura. Por ejemplo, se podría optimizar la búsqueda mediante estructuras indexadas o implementar algoritmos más eficientes cuando se manejen listas de gran tamaño. En conjunto, este proyecto demuestra cómo un algoritmo básico como la búsqueda secuencial puede integrarse de forma efectiva dentro de una aplicación real, manteniendo la robustez, legibilidad y facilidad de uso del sistema.

V. Análisis a Priori

Bubble Sort

1. Eficiencia espacial

   Utiliza una lista doblemente enlazada existente. No requiere estructuras auxiliares adicionales ni crea copias de la lista. Solo usa variables temporales para intercambiar los valores de los nodos.

   ➔ Espacio adicional: O(1)

2. Eficiencia temporal

   Compara cada par de nodos múltiples veces. En cada pasada, realiza comparaciones e intercambios entre elementos vecinos hasta que la lista esté ordenada.

➔ Recorre la lista n veces, y en cada pasada compara hasta n-1 elementos.

3. Análisis de Orden

   - Peor caso: $O(n^2)$
   - Promedio: $O(n^2)$
   - Mejor caso (lista ya ordenada con optimización de bandera): O(n)

Busqueda Secuencial

1. Eficiencia espacial

   Utiliza la lista existente de canciones cargada en memoria. No requiere estructuras auxiliares adicionales ni crea copias de la lista original. Solo emplea variables temporales para almacenar el dato a buscar y controlar el recorrido.

   Espacio adicional: O(1)

2. Eficiencia temporal

   Compara cada elemento de la lista uno por uno con el título buscado hasta encontrar coincidencia o agotar la lista. Cada comparación se realiza de forma secuencial y no se reutilizan resultados previos.

   Número de comparaciones: entre 1 y n, dependiendo de la posición del elemento.

3. Análisis de Orden

   Peor caso: O(n)

   Promedio: O(n)

   Mejor caso (coincidencia en la primera posición): O(1)

VI. Análisis a Posteriori

Bubble Sort

1. Análisis del mejor caso

    - Lista ya ordenada por prioridad y fecha límite.
    - La bandera cambio se mantiene en False después de la primera pasada.
    - ➔ Solo una pasada, sin intercambios.
    - ➔ Rendimiento real: muy eficiente, tiempo lineal (O(n))

2. Análisis del caso promedio

    - Lista con elementos en orden aleatorio.
    - Realiza múltiples pasadas con múltiples intercambios.
    - ➔ Rendimiento real: lento para listas grandes

3. Análisis del peor caso

    - Lista ordenada en orden inverso (peor posible).
    - Se necesitan n-1 pasadas con intercambios en cada una.
    - ➔ Rendimiento real: muy ineficiente ($O(n^2)$)

Busqueda Secuencial

Análisis del mejor caso

La canción buscada está ubicada en la primera posición de la playlist.

Solo se realiza una comparación y no es necesario recorrer más elementos.

Rendimiento real: muy eficiente, tiempo constante (O(1))

Análisis del caso promedio

La canción se encuentra en una posición intermedia dentro de la lista.

Se recorren varios elementos hasta encontrar coincidencia, realizando aproximadamente n/2 comparaciones.

Rendimiento real: adecuado para listas pequeñas, pero más lento en listas grandes.

Análisis del peor caso

La canción buscada está en la última posición o no existe en la playlist.

Se recorren todos los elementos de la lista sin coincidencia y se realizan n comparaciones.

Rendimiento real: menos eficiente, tiempo lineal completo (O(n))

VII. Resultados

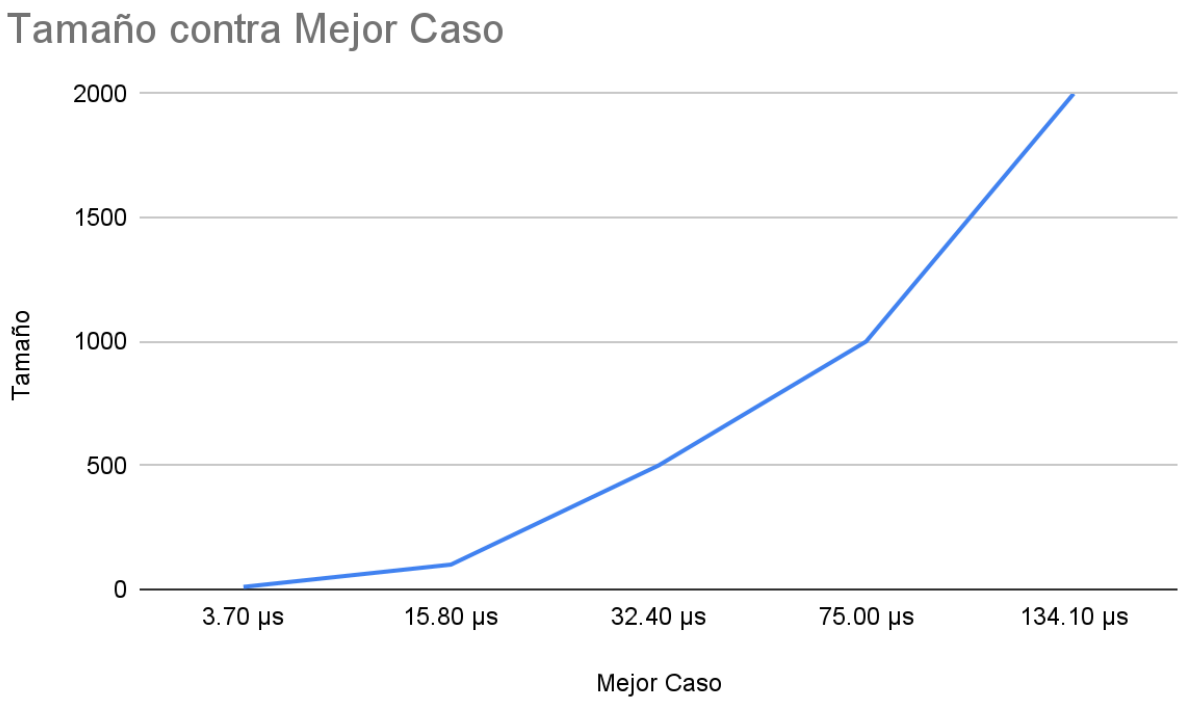
Bubble Sort:

La siguiente tabla presenta los tiempos de ejecución del algoritmo Bubble Sort evaluado en tres contextos: mejor caso (lista previamente ordenada), caso promedio (lista desordenada aleatoriamente) y peor caso (lista ordenada en sentido inverso). Los tiempos se midieron en diferentes tamaños de entrada para observar su comportamiento en escenarios reales.

Tamaño	Mejor Caso	Promedio	Peor Caso
10	3.70 µs	9.80 µs	7.00 µs
100	15.80 µs	1.09 ms	582.20 µs
500	32.40 µs	18.08 ms	17.49 ms
1000	75.00 µs	71.70 ms	63.53 ms
2000	134.10 µs	329.98 ms	266.61 ms

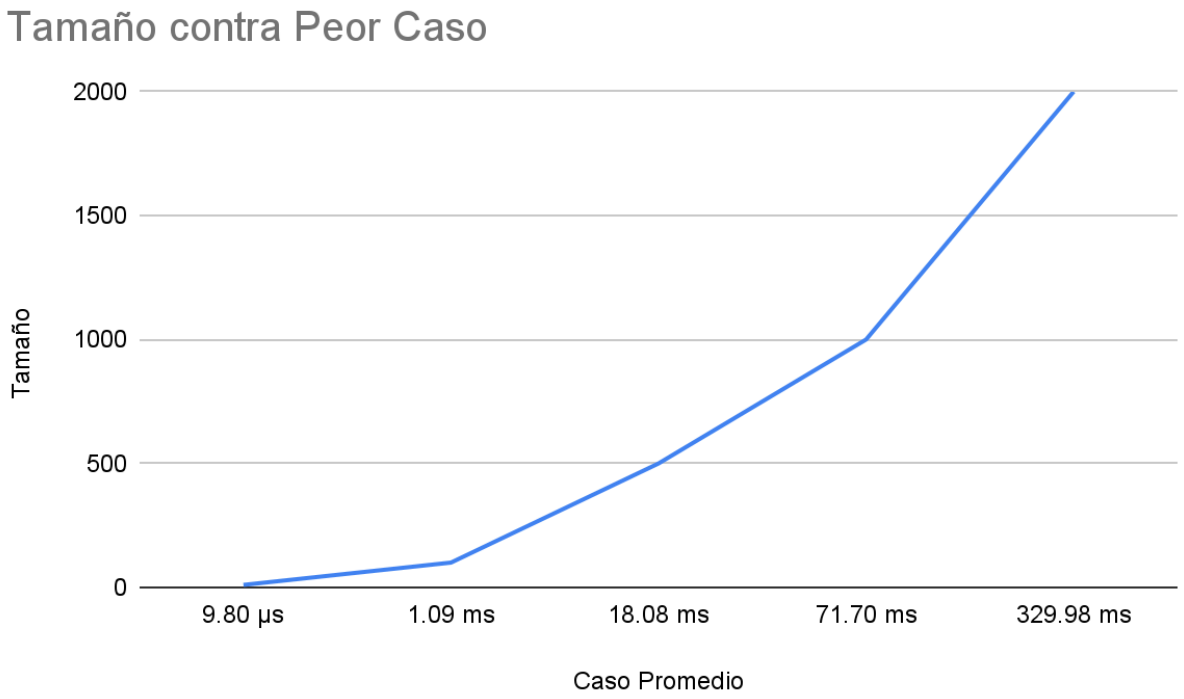
Mejor Caso:

En el mejor de los escenarios, cuando la lista ya se encuentra ordenada por prioridad y fecha límite, los tiempos de ejecución son significativamente bajos. Por ejemplo, para una lista de 10 elementos, el algoritmo tardó tan solo 3.70 µs, y para una de 2000 elementos, 134.10 µs. Este rendimiento se debe al uso del indicador de cambio (bandera) dentro de la implementación, que permite detectar que no es necesario realizar intercambios y finalizar la ejecución tras una sola pasada. El crecimiento observado es prácticamente lineal, lo cual concuerda con la complejidad teórica O(n) en este caso.



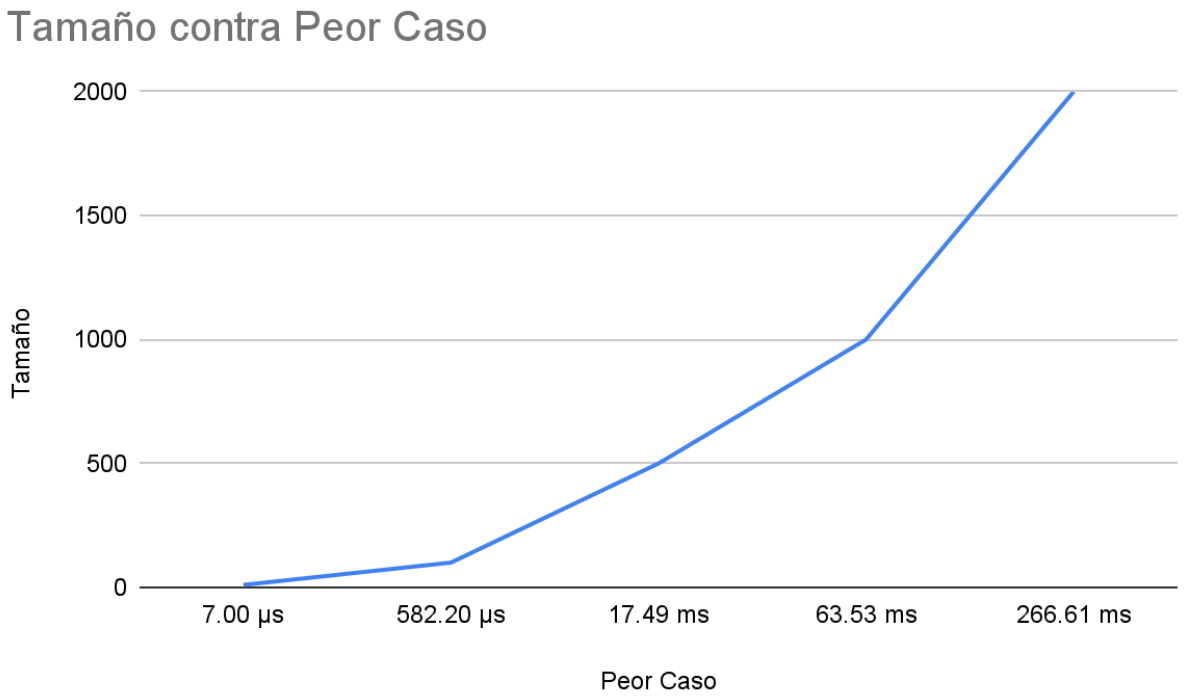
Caso Promedio:

En condiciones típicas con listas de datos desordenados de forma aleatoria, el algoritmo presenta un crecimiento de tiempo mucho más pronunciado. Para 100 elementos se registraron 1.09 ms, mientras que para 2000 elementos el tiempo ascendió a 329.98 ms. Estos resultados evidencian el impacto de los múltiples intercambios que Bubble Sort realiza en este contexto. El patrón de crecimiento temporal es cuadrático, reflejando una complejidad empírica $O(n^2)$. Se observa un aumento considerable en cada salto de tamaño, lo que pone de manifiesto la inestabilidad del algoritmo en contextos no favorables.



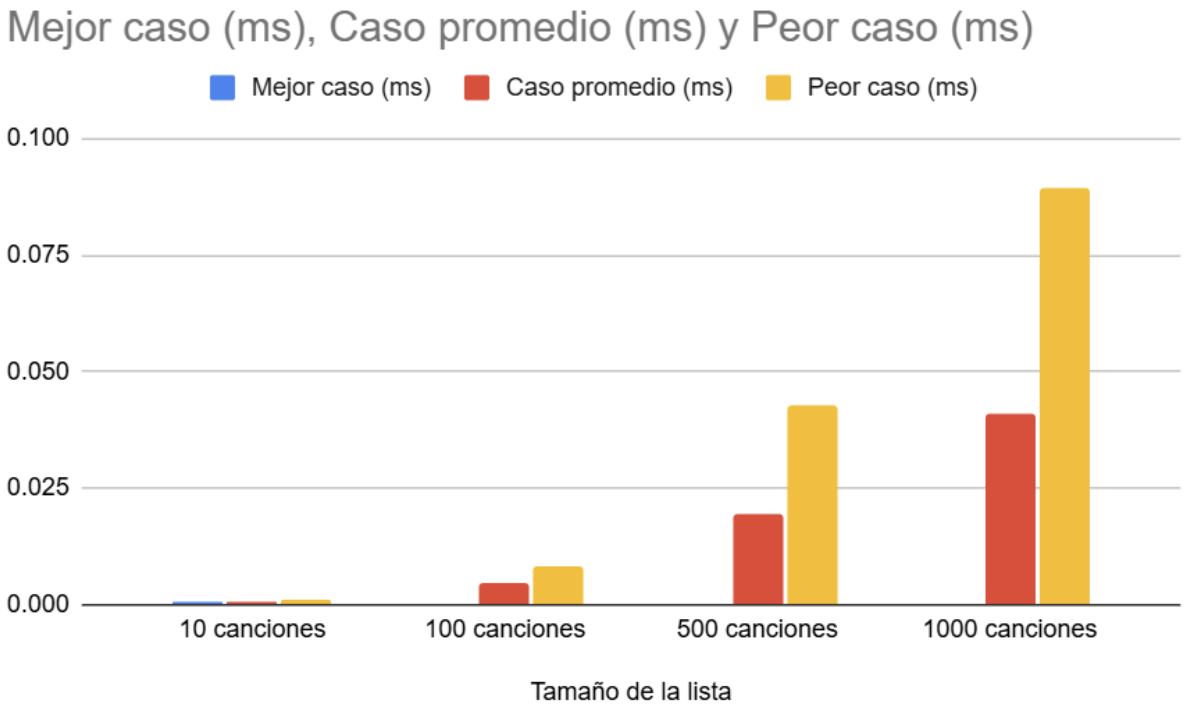
Peor Caso:

El peor caso, correspondiente a listas ordenadas de forma completamente inversa, también refleja un comportamiento de crecimiento cuadrático, aunque con ciertas variaciones respecto al caso promedio. Por ejemplo, para 1000 elementos se registró un tiempo de 63.53 ms, y para 2000 elementos 266.61 ms. El número de operaciones de comparación e intercambio es máximo en este caso, ya que cada elemento debe desplazarse al extremo opuesto. No obstante, es importante notar que el tiempo en este caso no supera al del caso promedio para todos los tamaños, lo cual podría deberse a variaciones del entorno de ejecución (procesador, caché, carga del sistema, etc.). Aun así, el comportamiento global respalda la complejidad $O(n^2)$ predicha teóricamente.



Búsqueda Secuencial

Tamaño de la lista	Mejor caso (ms)	Caso promedio (ms)	Peor caso (ms)
10 canciones	0.00059	0.00064	0.001
100 canciones	0.00026	0.00463	0.00832
500 canciones	0.00026	0.01932	0.04274
1000 canciones	0.0003	0.04106	0.08934



Análisis cuantitativo de los resultados de búsqueda secuencial

Los datos experimentales obtenidos mediante la medición del tiempo de ejecución del algoritmo de búsqueda secuencial reflejan un comportamiento acorde con la teoría esperada para este tipo de algoritmo. Se evaluaron tres casos representativos: el mejor caso (cuando la canción buscada está en la primera posición), el caso promedio (cuando la canción se encuentra aproximadamente a la mitad de la lista) y el peor caso (cuando la canción no se encuentra o está al final de la lista).

Como se observa en la tabla y las gráficas, el tiempo de búsqueda incrementa conforme aumenta el tamaño de la playlist. En el mejor caso, el tiempo se mantiene prácticamente constante y muy bajo, debido a que el algoritmo termina su ejecución tras la primera comparación exitosa, reflejando un comportamiento de tiempo constante O(1). Esto queda evidenciado con tiempos mínimos incluso para listas grandes, confirmando la eficiencia en esta condición ideal.

En el caso promedio, donde la búsqueda recorre aproximadamente la mitad de los elementos, el tiempo de ejecución aumenta linealmente con el tamaño de la lista. Los resultados muestran que, para listas pequeñas, el incremento es poco significativo, pero para tamaños mayores como 500 o 1000 canciones, la latencia se hace más notable. Esto es coherente con la complejidad temporal O(n) que caracteriza a la búsqueda secuencial.

El peor caso representa la condición donde el algoritmo debe revisar todos los elementos sin encontrar la canción buscada. En este escenario, los tiempos son los más altos y aumentan proporcionalmente al tamaño de la lista, corroborando la complejidad O(n) en tiempo lineal. La diferencia entre el caso promedio y el peor caso es moderada, pero sigue una tendencia similar.

VIII. Conclusiones

La presente investigación permitió analizar de manera integral la eficiencia de dos algoritmos fundamentales: Bubble Sort para ordenamiento y Búsqueda Secuencial para localización de elementos en arreglos desordenados. A través del análisis teórico (a priori) y de la experimentación práctica (a posteriori), se logró evaluar el comportamiento de ambos algoritmos en distintos escenarios de entrada y bajo diferentes condiciones de ejecución.

Los resultados obtenidos evidencian que Bubble Sort, si bien resulta intuitivo y de fácil implementación, presenta importantes limitaciones en términos de eficiencia temporal, especialmente en el caso promedio y el peor caso, donde su complejidad cuadrática ($O(n^2)$) lo hace poco viable para listas de tamaño considerable. No obstante, su rendimiento mejora significativamente cuando la lista ya está ordenada o casi ordenada, gracias al uso de la optimización con bandera, lo cual reduce su comportamiento a tiempo lineal en el mejor caso ($O(n)$).

Por otro lado, la Búsqueda Secuencial demostró ser adecuada únicamente en contextos donde se trabaja con listas pequeñas o sin orden preestablecido, dado que revisa uno a uno los elementos hasta encontrar el valor buscado. Su principal ventaja radica en su simplicidad y en no requerir una lista previamente ordenada. Sin embargo, su eficiencia también decrece de forma proporcional al crecimiento del tamaño de los datos, siendo su rendimiento en el peor caso igualmente lineal ($O(n)$).

Entre los hallazgos más importantes de este estudio se destacan:

- La eficiencia de un algoritmo depende no solo de su estructura interna, sino también del tipo y orden de los datos de entrada.
- Bubble Sort, pese a su ineficiencia general, puede ser útil en contextos educativos o cuando se requiere un algoritmo sencillo para listas pequeñas.
- La Búsqueda Secuencial resulta práctica en situaciones donde la búsqueda es eventual y los datos no justifican el uso de algoritmos más complejos o estructuras ordenadas.
- Ambos algoritmos pueden considerarse como herramientas pedagógicas útiles para introducir conceptos de complejidad y análisis algorítmico.

Como recomendación general, se sugiere evitar el uso de Bubble Sort y Búsqueda Secuencial en aplicaciones que manejen grandes volúmenes de datos o donde el rendimiento sea crítico. En tales casos, es preferible optar por algoritmos de ordenamiento más eficientes como Quick Sort o Merge Sort, y estructuras de búsqueda como árboles binarios o tablas hash.

Finalmente, esta investigación permitió sentar una base sólida para la comprensión de conceptos fundamentales relacionados con la eficiencia computacional, la importancia del análisis de casos, y el impacto que puede tener la elección de un algoritmo en el desempeño general de un sistema.

IX. Referencias bibliográficas

Campos, F. (2024, 19 de abril). *Bubble Sort: Lo que debes saber para tu próxima entrevista*. LinkedIn. https://es.linkedin.com/pulse/bubble-sort-lo-que-debes-saber-para-tu-pr%C3%B3xima-fernando-campos-anegf

Canquil, A. (s.f.). *Unidad I: Eficiencia de algoritmos*. https://canquil.wordpress.com/unidad-i/

CertiDevs. (2025, 24 de enero). *Búsqueda Secuencial: Implementación y Análisis*. https://certidevs.com/tutorial-fundamentos-programacion-busqueda-algoritmos

DataCamp. (s.f.). *Notación Big O y guía de complejidad temporal*. https://www.datacamp.com/es/tutorial/big-o-notation-time-complexity

Dialgo. (2012, 23 de mayo). *Análisis a priori y prueba a posteriori*. http://di-algo-monica.blogspot.com/2012/05/analisis-priori-y-prueba-posteriori_23.html

FasterCapital. (2025, 25 de abril). *Clasificación de burbujas: análisis de pasos de clasificación secuencial*. https://fastercapital.com/es/contenido/Clasificacion-de-burbujas--analisis-de-pasos-de-clasificacion-secuencial.html

Guano, A. (s.f.). *Complejidad espacial*. Platzi. https://platzi.com/cursos/complejidad-js/complejidad-espacial/

Olawanle, J. (2024, 1 de febrero). *Guía: Notación Big O - Gráfico de complejidad de tiempo*. freeCodeCamp. https://www.freecodecamp.org/espanol/news/hoja-de-trucos-big-o/

Sunter, B. (2022, 22 de agosto). *Posteriori vs A Priori Analysis of Algorithms*. https://briansunter.com/posteriori-vs-a-priori-analysis-of-algorithms