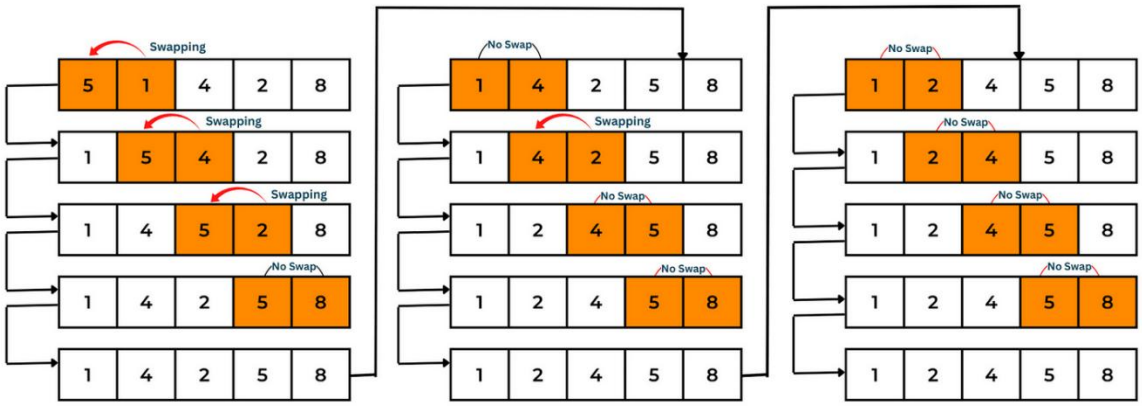
Trabajo Practico Integrador

“Algoritmos de Ordenamiento y Búsqueda”



ALUMNOS:

DA CORTÁ, Manuel – manueldacorta@gmail.com

ECHEVERRIA ARAYA, Agustin – agustinecheverria2019@gmail.com

MATERIA: Programación I – Comisión II

PROFESORES: Julieta Trapé – Miguel Barrera Oltra

09/06/25

Contenido

1. Introducción

En la era actual de la información, donde el volumen de datos crece exponencialmente, la gestión eficiente de la información se ha convertido en un pilar fundamental de la programación y la ciencia de la computación. Los algoritmos de ordenamiento y búsqueda son herramientas esenciales que permiten organizar y localizar datos de manera efectiva, optimizando el rendimiento de las aplicaciones y sistemas. Comprender su funcionamiento, sus complejidades y sus aplicaciones es crucial para el desarrollo de software robusto y eficiente.

El presente trabajo se propone explorar en profundidad estos algoritmos. Se analizarán distintos métodos de ordenamiento como el Bubble Sort, Selection Sort e Insertion Sort, destacando sus características, ventajas y desventajas en términos de rendimiento. Posteriormente, se abordarán los algoritmos de búsqueda, haciendo un énfasis especial en la búsqueda binaria, dada su relevancia y eficiencia en estructuras de datos ordenadas.

A través de este estudio, no solo se busca comprender la teoría detrás de estos algoritmos, sino también su aplicación práctica. Se establecerán criterios para la elección del algoritmo más adecuado según el contexto y las características de los datos, demostrando la importancia de un diseño algorítmico consciente para la optimización de procesos.

2. Marco Teórico

2.1. Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son procedimientos computacionales diseñados para reorganizar los elementos de una colección (como una lista o un arreglo) en una secuencia específica, ya sea ascendente o descendente. El ordenamiento es una operación preparatoria fundamental que mejora drásticamente la eficiencia de operaciones posteriores, especialmente la búsqueda, facilitando el análisis, la visualización y el procesamiento de los datos.

- **Bubble Sort (Ordenamiento de Burbuja):**

  Este algoritmo es uno de los más simples de entender e implementar. Funciona comparando repetidamente pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Con cada pasada, los

elementos "más grandes" (o "más pequeños", según el criterio de ordenación) "ascienden" a su posición final, como burbujas en el agua.

- **Rendimiento:** Su complejidad temporal es O(n2) en el peor y promedio caso, lo que lo hace ineficiente para grandes conjuntos de datos debido al elevado número de comparaciones e intercambios.

- **Selection Sort (Ordenamiento por Selección)**

En cada iteración, el algoritmo de selección busca el elemento más pequeño (o más grande, según el orden deseado) del resto de la lista no ordenada y lo coloca en la posición correcta al principio de la parte no ordenada. Aunque realiza menos intercambios que Bubble Sort, la búsqueda del elemento mínimo en cada paso mantiene su rendimiento.

- **Rendimiento:** Su complejidad temporal es O(n2) en todos los casos (mejor, promedio y peor).

- **Insertion Sort (Ordenamiento por Inserción)**

Este algoritmo construye la lista ordenada de 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. Es particularmente eficiente para conjuntos de datos pequeños o listas que ya están parcial o casi completamente ordenadas.

- **Rendimiento:** Su complejidad temporal es O(n2) en el peor y promedio caso, pero O(n) en el mejor caso (cuando la lista ya está ordenada), lo que lo convierte en una opción viable en escenarios específicos.

Consideraciones Generales en el Ordenamiento: La elección de un algoritmo de ordenamiento va más allá de la complejidad temporal. Factores como la estabilidad (si mantiene el orden relativo de elementos con valores iguales), el uso de memoria (si requiere memoria adicional o si opera "in-place") y la complejidad de implementación son cruciales para determinar su idoneidad en diferentes escenarios. Algoritmos como Mergesort o Quicksort ofrecen una complejidad de O(nlogn), siendo significativamente más eficientes para grandes volúmenes de datos, aunque su implementación es más compleja.

2.2. Algoritmos de Búsqueda

Los algoritmos de búsqueda tienen como objetivo principal localizar un valor específico dentro de una colección de elementos. La eficiencia de estos algoritmos es de suma importancia, especialmente cuando se trabaja con grandes volúmenes de datos donde una búsqueda ineficiente puede impactar gravemente el rendimiento del sistema.

- **Búsqueda Lineal (Secuencial):** Este es el método de búsqueda más directo y simple. Consiste en examinar cada elemento de la lista secuencialmente, uno por uno, hasta que se encuentra el elemento objetivo o hasta que se ha recorrido toda la lista.

    - **Rendimiento:** Su complejidad temporal es O(n) en el peor y promedio caso, ya que, en el peor escenario, el elemento buscado podría ser el último o no estar presente en la lista, requiriendo el recorrido completo de los n elementos.

    - **Características:** Es aplicable a cualquier tipo de lista, independientemente de si está ordenada o desordenada, lo que lo hace versátil pero ineficiente para colecciones extensas.

- **Búsqueda Binaria:** La búsqueda binaria es un algoritmo altamente eficiente para encontrar un elemento en una *lista que debe estar previamente ordenada*. Su funcionamiento se basa en un enfoque de "divide y vencerás". El algoritmo comienza comparando el elemento objetivo con el valor medio de la lista. Si el valor medio es el objetivo, la búsqueda finaliza. Si no, la búsqueda se reduce a la mitad superior o inferior de la lista, descartando la otra mitad. Este proceso se repite, dividiendo el espacio de búsqueda por la mitad en cada iteración, hasta que el elemento es encontrado o se determina que no está presente.

    - **Precondición:** La condición *sine qua non* para la búsqueda binaria es que la colección de datos debe estar **ordenada**. Sin esta pre condición, el algoritmo no garantiza resultados correctos.

    - **Rendimiento:** Su complejidad temporal es O(logn). Esta eficiencia logarítmica la hace exponencialmente más rápida que la búsqueda lineal para grandes conjuntos de datos ordenados, ya que el número de operaciones crece de forma muy lenta con el tamaño de la entrada.

- **Implementación:** Puede ser implementada de forma iterativa (usando bucles) o recursiva (definiendo una función que se llama a sí misma). Ambas versiones producen resultados idénticos y confiables, demostrando su efectividad. La versión iterativa es sencilla de escribir, fácil de depurar, eficiente en el uso de memoria y estable, sin riesgo de desbordamiento de pila. La versión recursiva, aunque matemáticamente más elegante e intuitiva, puede consumir más memoria debido a las llamadas recursivas, lo que podría resultar en un rendimiento ligeramente inferior en algunos casos

2.3. Comparación y Elección de Algoritmos

La selección del algoritmo de búsqueda o ordenamiento más adecuado es una decisión estratégica que depende de varios factores contextuales y de rendimiento:

- **Tamaño del conjunto de datos**: Para conjuntos de datos pequeños, la diferencia de rendimiento entre algoritmos de diferentes complejidades (O(n2) vs. O(nlogn) o O(logn)) puede ser insignificante. Sin embargo, para volúmenes de datos grandes, la elección de algoritmos con menor complejidad asintótica se vuelve crucial para garantizar la eficiencia y escalabilidad del sistema.

- **Estado de los datos:** Si los datos ya están ordenados o se espera que lo estén (por ejemplo, en bases de datos indexadas), algoritmos como la Búsqueda Binaria son la opción óptima. Si los datos están desordenados y el ordenamiento no es una precondición para otras operaciones, la Búsqueda Lineal podría ser suficiente o el costo de ordenar superaría el beneficio de una búsqueda más rápida.

- **Necesidad de estabilidad (solo ordenamiento):** Para escenarios donde el orden relativo de elementos con valores iguales debe preservarse, es fundamental seleccionar un algoritmo de ordenamiento estable (ej., Insertion Sort, Merge Sort).

- **Uso de memoria (Espacio):** Algunos algoritmos requieren memoria adicional para su operación (ej., Merge Sort), mientras que otros operan "in-place", minimizando el uso de memoria auxiliar (ej., Bubble Sort, Quicksort). Esta consideración es importante en entornos con restricciones de memoria.

- **Complejidad de implementación:** La facilidad o dificultad de codificar y mantener un algoritmo también influye en su elección, especialmente en proyectos con plazos ajustados o equipos de desarrollo con diferentes niveles de experiencia.

La eficiencia de los algoritmos se analiza utilizando la notación O (como O(n), O(log n), O(n^2)), considerando el peor caso, el mejor caso y el caso promedio.

2.4. Aplicaciones Prácticas de la Búsqueda Binaria

La búsqueda binaria, debido a su eficiencia, tiene numerosas aplicaciones en la vida real:

- **Bases de datos indexadas:** Para búsquedas eficientes en bases de datos relacionales.
- **Sistemas de archivo:** Para buscar elementos en sistemas operativos y sistemas de almacenamiento.
- **Aplicaciones científicas:** En el procesamiento de información científica, estadísticas y modelos de machine learning donde se manejan grandes conjuntos de datos.

La búsqueda binaria es mucho más rápida y escalable para listas grandes en comparación con la búsqueda lineal.

3. Caso Practico

Para aplicar los conceptos teóricos estudiados, se desarrolló un programa en Python que permite:

- Ordenar una lista de números ingresados por el usuario utilizando el algoritmo Insertion Sort.

- Buscar un número dentro de esa lista mediante el algoritmo de Búsqueda Binaria.

Se eligió Insertion Sort por su simplicidad de implementación y buen rendimiento en listas pequeñas o parcialmente ordenadas. La Búsqueda Binaria, por su parte, fue seleccionada por su eficiencia en listas ordenadas, con una complejidad logarítmica.

```python
# Algoritmo de ordenamiento: Insertion Sort
def insertion_sort(lista):
    for i in range(1, len(lista)):
        actual = lista[i]
        j = i - 1
        while j >= 0 and actual < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual

# Algoritmo de búsqueda: Búsqueda Binaria
def busqueda_binaria(lista, objetivo):
    izquierda = 0
    derecha = len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# Lista de prueba
numeros = [42, 7, 19, 3, 25, 18, 12]
print("Lista original:", numeros)

# Ordenar la lista
insertion_sort(numeros)
print("Lista ordenada:", numeros)

# Buscar un número
numero_a_buscar = 19
posicion = busqueda_binaria(numeros, numero_a_buscar)

# Mostrar resultados
if posicion != -1:
    print(f"El número {numero_a_buscar} se encuentra en la posición {posicion}.")
else:
    print(f"El número {numero_a_buscar} no se encuentra en la lista.")
```

4. Metodología Utilizada

La elaboración del trabajo se realizó en las siguientes etapas:

- **Revisión teórica:** se comenzó el trabajo leyendo y analizando el material brindado por la cátedra, para comprender el funcionamiento y las características principales de los algoritmos de búsqueda y ordenamiento.

- **Selección de algoritmos:** se eligieron Insertion Sort y Búsqueda Binaria por su simplicidad, claridad lógica y adecuación al nivel inicial de programación, además de su integración práctica (la búsqueda binaria requiere una lista ordenada).

- **Desarrollo del código:** se implementaron ambos algoritmos en lenguaje Python, cuidando que el código fuera claro, legible y estuviera correctamente indentado. Se agregaron comentarios explicativos en las secciones clave para facilitar su interpretación.

- **Verificación del funcionamiento:** se probó el programa con ejemplos simples para asegurar que cumpliera con los objetivos planteados, verificando que ordenara correctamente la lista y que la búsqueda arrojara resultados coherentes.

5. Resultados Obtenidos

Se logró desarrollar un programa funcional que permite ordenar listas numéricas mediante el algoritmo de Insertion Sort, y aplicar sobre ellas el algoritmo de Búsqueda Binaria para encontrar un número determinado. El código permitió visualizar de forma concreta cómo la búsqueda binaria requiere que los datos estén previamente ordenados para funcionar correctamente, lo cual se cumplió con éxito al utilizar el algoritmo de ordenamiento.

El programa demostró un comportamiento esperado en todos los casos planteados, arrojando resultados coherentes y precisos. Se confirmó que ambos algoritmos son adecuados para trabajar con listas pequeñas, como las utilizadas a modo de ejemplo en este trabajo.

6. Conclusiones

Este trabajo permitió comprender en profundidad cómo funcionan los algoritmos de búsqueda y ordenamiento, tanto en su lógica como en su implementación práctica en Python. Se destacó la importancia de tener listas ordenadas para aprovechar algoritmos eficientes como la búsqueda binaria.

Además, se aprendió a evaluar qué algoritmo conviene usar según el tamaño y estado de los datos. El desarrollo colaborativo facilitó el aprendizaje y permitió integrar teoría y práctica en un entorno de programación real. Como mejora futura, se podría comparar el rendimiento entre varios algoritmos utilizando herramientas de medición de tiempo.

7. Bibliografía

Universidad Tecnológica Nacional (UTN). (2025). *Material provisto por la catedra sobre Algoritmos de Ordenamiento y Búsqueda*. Tecnicatura Universitaria en Programación a Distancia (TUPaD).