Algoritmos de Búsqueda y Ordenamiento

Trabajo Práctico Integrador

Materia: Programación 1
Profesor: Cinthia Rigoni
Tutor: Oscar Londero

Alumnos:

- Tomás Agustín Benítez – agustinbenitez159@gmail.com
- Tomás Esteban Bohorquez – estebanbohorquez86@gmail.com

Fecha de entrega: 9 de junio de 2025

Índice

Introducción

En el campo de la informática y el desarrollo de software, el procesamiento eficiente de la información es un aspecto crítico. Tanto en sistemas pequeños como en infraestructuras de gran escala, la necesidad de buscar y organizar datos de forma rápida y eficiente es constante. Es aquí donde cobran importancia los algoritmos de búsqueda y ordenamiento, herramientas fundamentales para lograr que los programas gestionan los datos eficazmente y con precisión

Los algoritmos de búsqueda localizan elementos dentro de estructuras como puede ser una lista, una matriz, un árbol o una base de datos, mientras que los de ordenamiento reorganizan datos según un criterio. Ambos son esenciales porque, además de su función principal, optimizan procesos, como la búsqueda binaria (requiere una estructura ordenada), el análisis de datos, la compresión de archivos y el almacenamiento eficiente

Este trabajo práctico tiene como finalidad no solo repasar los principales algoritmos de búsqueda y ordenamiento, sino también analizar sus características, implementar versiones propias en lenguaje Python y evaluar de forma práctica su rendimiento. Se pretende que, al finalizar este trabajo, se comprenda tanto desde el punto de vista teórico como práctico cual es el comportamiento esperado de cada algoritmo, en qué contextos es conveniente aplicarlos y cuales son sus limitaciones.

Dado que la cantidad de algoritmos tanto de ordenamiento como de búsqueda es demasiado amplio y diverso, en el trabajo se ha optado por seleccionar y desarrollar tres algoritmos representativos de búsqueda y tres de ordenamiento, elegidos por su relevancia y valor didáctico.

Marco Teórico

Algoritmos de Búsqueda

En ciencias de la computación, un algoritmo de búsqueda es un procedimiento usado para encontrar un elemento específico dentro de una colección de datos, como una lista, un arreglo, una base de datos o incluso estructuras más complejas como árboles o grafos. Su importancia radica en que gran parte de las tareas que realiza una computadora implican localizar información dentro de grandes volúmenes de datos.

Los algoritmos de búsqueda se dividen generalmente en 2 grandes categorías: búsqueda secuencial y búsqueda no secuencial. En este trabajo se abordarán tres algoritmos fundamentales que ilustran distintas estrategias de búsqueda.
Para facilitar su análisis, los organizamos en orden creciente de complejidad: búsqueda lineal, búsqueda binaria y Jump Search. Cada uno responde a necesidades y escenarios distintos, y su correcta elección depende de comprender cómo trabajan internamente.

Uno de los criterios más importantes al analizar estos algoritmos es su eficiencia, tanto en término de tiempo de ejecución como de recursos utilizados. Esta eficiencia se expresa generalmente en notación Big O, que permite comparar el rendimiento de los algoritmos según el tamaño de los datos.

Además del análisis de eficiencia, también se consideran factores como la estructura de datos en la que se realiza la búsqueda (por ejemplo, arreglos ordenados o no ordenados) y la probabilidad de encontrar o no el elemento buscado.

Búsqueda Lineal

La búsqueda lineal (también conocida como búsqueda secuencial) es el algoritmo más simple y directo. Consiste en recorrer cada elemento del conjunto, uno por uno, hasta encontrar el valor buscado o llegar al final de la colección.

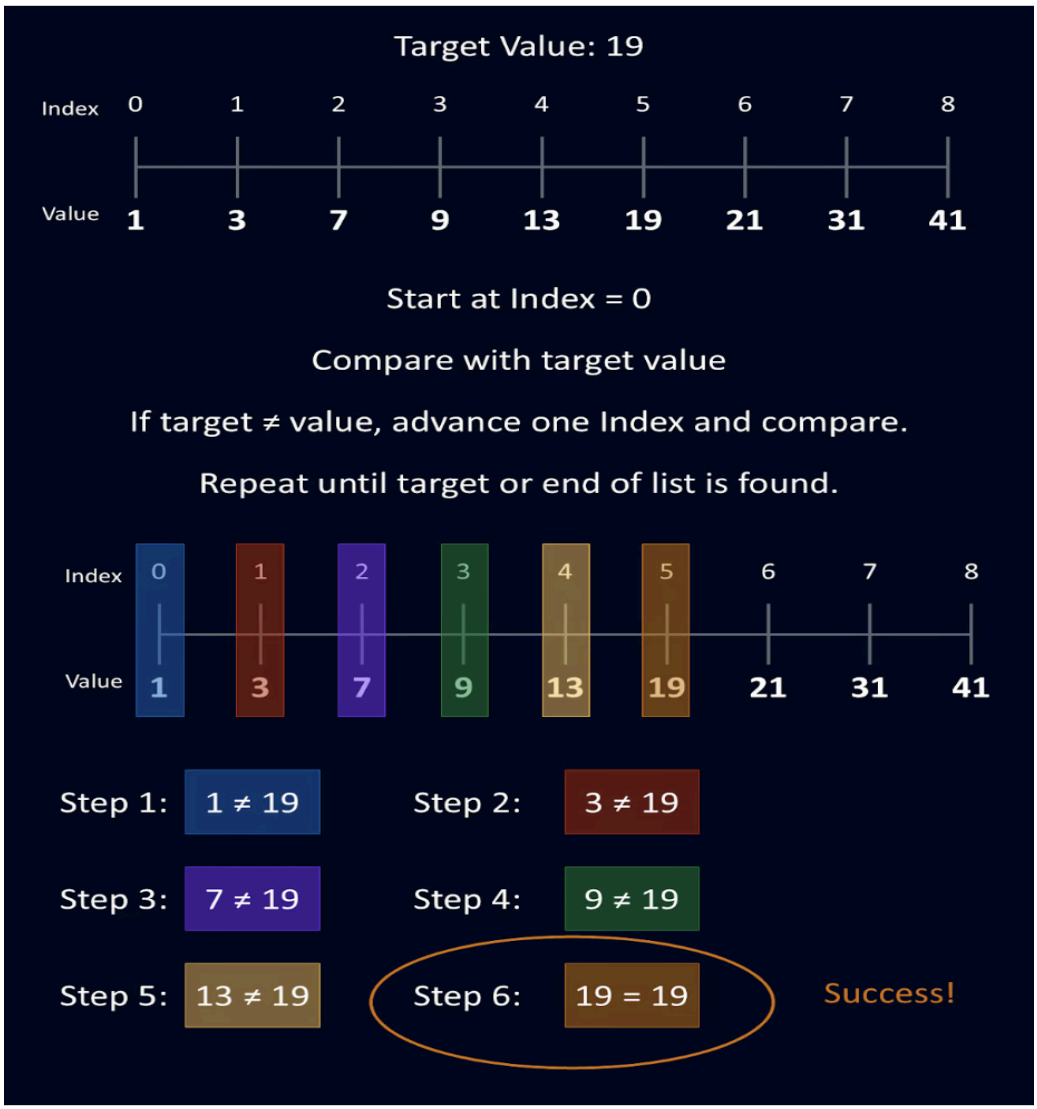
Un ejemplo, que puede reflejarse en la vida cotidiana, es por ejemplo que se quiere buscar una remera específica en un cajón lleno de ropa. No está ordenado de ninguna manera, así que tenes que ir sacando

una prenda por vez y mirar si es la remera que elegiste. Si no es, seguís con la siguiente, y así hasta que encontrás la que querías o se te haya acabado la ropa.
Complejidad del algoritmo:

- **Mejor caso:** en el mejor caso, la clave podría estar presente en el primer índice. Por lo tanto, la complejidad en el mejor caso es O(1).

- **Peor caso:** en el peor de los casos, la clave podría estar presente en el último índice, es decir, en el extremo opuesto al que se inició la búsqueda en la lista. Por lo tanto, la complejidad en el peor de los casos es O(N), donde N es el tamaño de la lista.

- **Caso promedio:** O(N)



¿En qué casos concretos conviene usar una búsqueda lineal, a pesar de no ser la más eficiente?

- **Cuando es clave la sencillez:** el algoritmo es fácil de entender y aplicar. No hay que preocuparse de complejas ordenaciones o divisiones de los datos. Simplemente empieza por el principio de la lista y comprueba cada elemento hasta que encuentres lo que buscas.
- **Cuando no hay tiempo para ordenar:** la búsqueda lineal no requiere que el conjunto de datos esté ordenado. Esto lo hace perfecto para cuando se necesita encontrar algo rápidamente.
- **Cuando se requiere versatilidad:** es lo suficientemente flexible como para manejar distintos tipos de datos, desde números a cadenas, e incluso objetos.

Búsqueda Binaria

A diferencia de la búsqueda lineal, que no requiere ningún tipo de orden, la búsqueda binaria exige que los datos estén previamente ordenados. Este cambio de enfoque permite ganar mucha eficiencia, pero también impone nuevas condiciones sobre la estructura de los datos.

Divide el conjunto a la mitad repetidamente, comparando el valor buscado con el valor medio actual. Esto reduce drásticamente la cantidad de comparaciones necesarias, haciendo al algoritmo muy eficiente.

Para ejemplificar con algo más cotidiano, la búsqueda binaria es similar a buscar una palabra en un diccionario. Como sabes que las palabras están ordenadas alfabéticamente, no arrancas desde la primera página: “partis” el diccionario al medio. Si la palabra que buscas va antes que la que aparece en esa página, buscas en la primera mitad del diccionario. Si va después, en la segunda mitad. Vas reduciendo el grupo de páginas a la mitad cada vez, hasta dar con la palabra exacta.

Complejidad del algoritmo:

- **Mejor caso:** O(1)
- **Peor caso:** O(log N)
- **Caso promedio:** O(log N)

Target Value: 31

Find middle index: (0 + 8) / 2 = 4

Compare target value with value at index 4: 13 < 31

Find middle index: (5 + 8) / 2 = 6 (rounded down)

Compare target value with value at index 6: 21 < 31

Find middle index: (7 + 8) / 2 = 7 (rounded down)

Compare target value with value at index 7: 31 = 31

SUCCESS! Target value is at Index 7.

¿Qué tipo de estructuras de datos permiten aplicar búsqueda binaria?

Se puede aplicar búsqueda binaria sobre cualquier estructura de datos que permita acceder a los elementos de forma directa (una lista, una cadena, etc), y que contenga datos previamente ordenados.

¿Qué pasaría si los datos no estuvieran ordenados?

Si los datos no están ordenados al aplicar la búsqueda binaria no puede aplicarse correctamente y podría no encontrar el elemento que se busca, o incluso devolver resultados erróneos. La búsqueda binaria depende de la capacidad de reducir el espacio de búsqueda a la mitad en cada paso, lo cual solo es posible con datos ordenados.

Jump Search

Finalmente, existe un enfoque intermedio que busca combinar lo mejor de ambos mundos: la simplicidad de la búsqueda lineal y la eficiencia de la binaria. Jump Search, o búsqueda por saltos, representa esa estrategia híbrida.
También requiere que los datos estén ordenados. En lugar de revisar elemento por elemento, salta en bloques fijos (por ejemplo, $\sqrt{n}$ pasos), hasta pasar el valor buscado. Luego, realiza una búsqueda lineal dentro del bloque donde pudo estar.

Por ejemplo, imaginemos que estás buscando un nombre en una lista impresa, ordenada alfabéticamente. En lugar de revisar uno por uno desde el inicio, lo más probable es que hagas un “salto” hasta la sección donde comienzan los nombres con la letra correspondiente. Una vez allí, recorren línea por línea hasta encontrar el nombre que buscás.

Complejidad del algoritmo:

- **Mejor caso:** $O(1)$
- **Peor caso:** $O(\sqrt{N})$
- **Caso promedio:** $O(\sqrt{N})$

¿Cómo se determina el tamaño ideal del salto?

El tamaño óptimo del salto en el algoritmo Jump Search se calcula como $\sqrt{n}$, donde n representa la cantidad total de elementos a buscar. Este valor permite equilibrar la cantidad de saltos con la cantidad de comparaciones dentro del bloque final, logrando una eficiencia razonable en la mayoría de los casos.

¿En qué escenarios jump search es preferible frente a búsqueda binaria?

- **Estructuras con acceso lento o costoso**: en listas enlazadas o datos almacenados en disco, donde moverse aleatoriamente es caro. Jump Search reduce los accesos aleatorios.

- **Sistemas donde se prioriza simplicidad y bajo consumo de memoria**: Jump Search no requiere recursión ni una pila como algunas implementaciones de búsqueda binaria.

- **Conjuntos de datos moderadamente grandes y estáticos**, donde se pueden calcular los bloques con anticipación y no es necesario reordenar ni modificar los datos frecuentemente.

¿Qué impacto tiene la elección del algoritmo de búsqueda en el rendimiento de sistemas complejos?

La elección del algoritmo de búsqueda es **crucial** en el diseño de sistemas a gran escala, ya que impacta directamente en el **rendimiento, la escalabilidad y la experiencia del usuario**. Cada sistema maneja volúmenes masivos de datos, y la eficiencia con la que se recupera información puede marcar la diferencia entre una respuesta instantánea y un cuello de botella crítico.

Para concluir, los algoritmos de búsqueda forman una parte esencial en la resolución de problemas computacionales. Desde métodos simples como la búsqueda lineal, hasta estrategias más elaboradas como la búsqueda binaria y Jump Search, cada uno responde a distintos contextos de uso. Comprender sus fundamentos teóricos no solo permite seleccionar el algoritmo adecuado, sino también diseñar soluciones más eficientes, robustas y adaptadas al problema real que se enfrenta.

Algoritmos de ordenamiento

Un algoritmo de ordenamiento tiene como objetivo reorganizar los elementos de una estructura de datos en un cierto orden, ya sea ascendente o descendente. Estos algoritmos son esenciales en diversas aplicaciones: desde la presentación ordenada de resultados en una interfaz gráfica hasta la optimización del almacenamiento y la mejora del rendimiento de otros algoritmos, como la búsqueda binaria.

Existen múltiples algoritmos de ordenamiento, cada uno con sus propias características, ventajas y desventajas. La elección del algoritmo correcto depende de factores como el tamaño de la estructura, si los datos ya están parcialmente ordenados, si se requiere estabilidad en el orden de los elementos, o si se trabaja con recursos limitados (como memoria o potencia de cómputo).

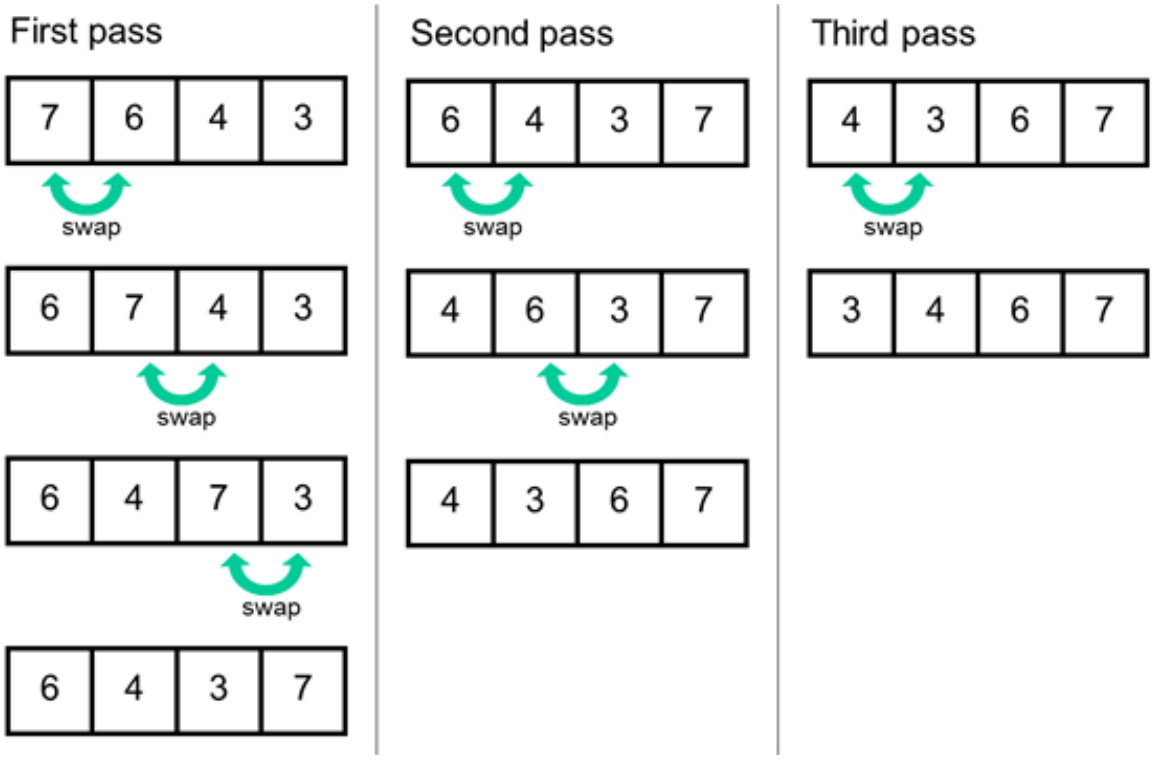
A continuación, se describen tres algoritmos de ordenamiento representativos: Bubble Sort, Merge Sort y Heap Sort seleccionados por su relevancia tanto en teoría como en aplicaciones reales.

Bubble Sort (Ordenamiento Burbuja)

Bubble Sort es un algoritmo de ordenamiento simple y fácil de entender, ideal para introducir el concepto de comparación e intercambio de elementos. Su funcionamiento consiste en recorrer repetidamente la lista, comparando elementos adyacentes e intercambiándose si están en el orden incorrecto. Este proceso se repite hasta que la lista esté completamente ordenada.

Durante cada pasada, los elementos más grandes se desplazan hacia el final de la lista. Aunque es poco eficiente para listas grandes, su simplicidad lo hace útil con fines educativos.

Este algoritmo es especialmente útil cuando se trabaja con listas pequeñas o casi ordenadas, o en entornos donde la simplicidad del código es prioritaria por encima del rendimiento.



Ventajas:

- Fácil de implementar: su lógica es directa y sencilla, ideal para estudiantes que recién comienzan.
- No requiere memoria adicional: es un algoritmo in-place, que ordena directamente sobre la lista original.
- Puede optimizarse ligeramente: se puede agregar una verificación para detenerse si en una pasada no se realizaron intercambios.

Desventajas:

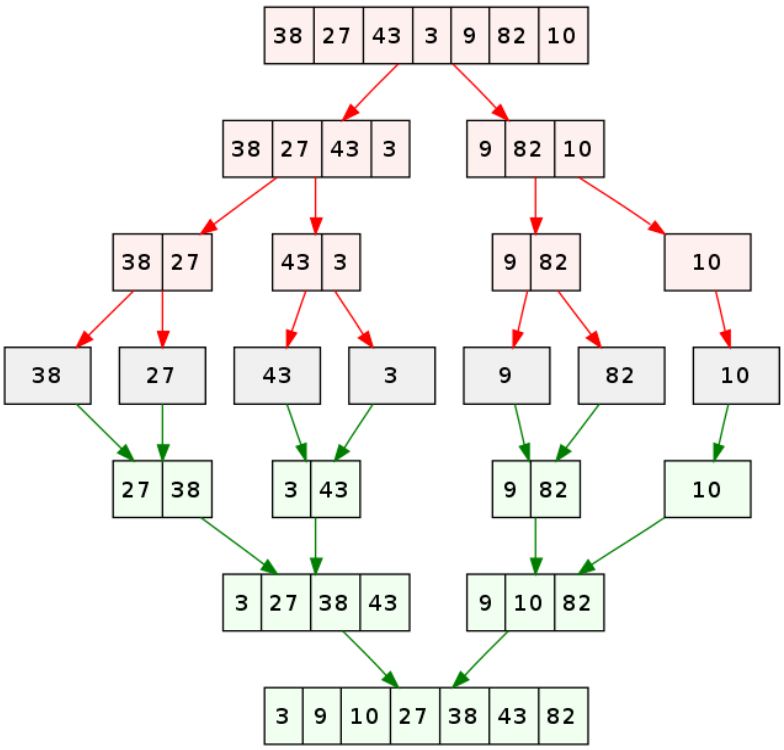
- Muy ineficiente para listas grandes: su rendimiento decrece rápidamente con el tamaño de la lista.
- No apto para aplicaciones que requieren rendimiento alto o procesamiento de grandes volúmenes de datos.

Complejidad:

- Peor caso: $O(n^2)$ *(cuando la lista está en orden inverso)*
- Mejor caso: O(n) (si ya está ordenada y se aplica una optimización que detecta que no hubo intercambios)
- Promedio: $O(n^2)$

Merge Sort (Ordenamiento por Mezcla)

Merge Sort es un algoritmo eficiente que aplica la estrategia de divide y vencerás. Su funcionamiento consiste en dividir la lista en mitades sucesivas hasta que cada sublista contenga un único elemento, para luego fusionarlas de forma ordenada en una lista mayor. Esta división y fusión se realiza recursivamente, garantizando un rendimiento estable y predecible de O(n log n), independientemente de la distribución inicial de los datos.



Este algoritmo es especialmente útil cuando se trabaja con grandes volúmenes de datos o con estructuras externas como archivos, debido a su estabilidad y rendimiento constante.

Ventajas:

- Estable: mantiene el orden relativo de los elementos iguales, lo cual es importante cuando el orden original tiene significado.

- Rendimiento garantizado de O(n log n) en todos los casos (mejor, peor y promedio).

- Muy eficiente para datos almacenados en medios externos (discos, archivos grandes).

Desventajas:

- Requiere memoria adicional proporcional al tamaño de la lista (O(n)), ya que no es un algoritmo in-place.

- Su uso de memoria extra puede ser un problema en sistemas con recursos limitados.

Complejidad:

- Peor caso: O(n log n)

- Mejor caso: O(n log n)

- Promedio: O(n log n)

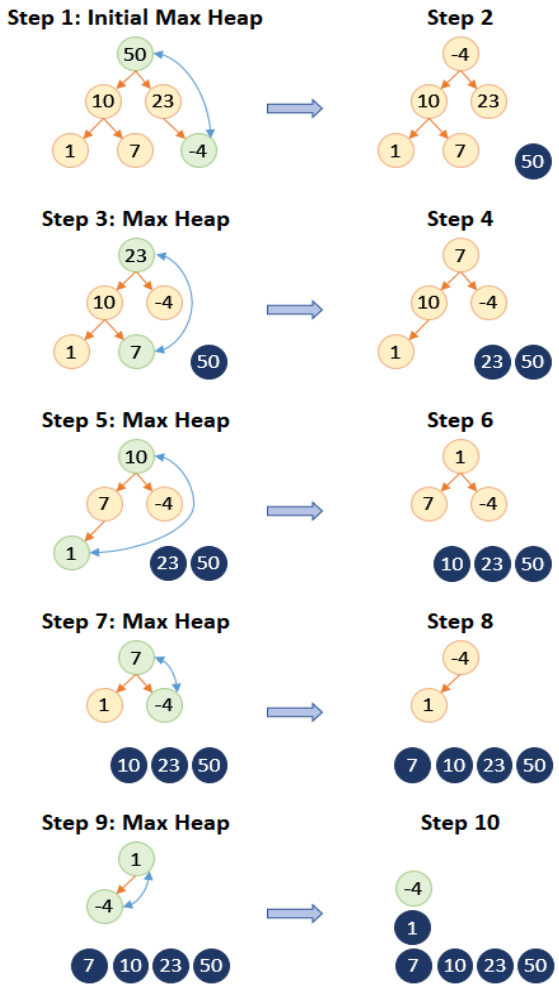
Heap Sort (Ordenamiento por Montículo)

Heap Sort utiliza una estructura de datos llamada **montículo** (heap), que es una estructura tipo árbol binario que puede representarse como un arreglo. El algoritmo comienza construyendo un montículo máximo o mínimo a partir de la lista desordenada. Luego, extrae repetidamente el elemento raíz (el mayor o menor, según el tipo de heap) y lo coloca al final de la lista, reduciendo el tamaño del heap progresivamente hasta ordenar toda la lista.

Este método se caracteriza por ser un algoritmo in-place, ya que no requiere memoria adicional significativa.

La estructura del heap sort se construye de arriba hacia abajo, de izquierda a derecha

Entonces, suponiendo que la lista es [50, 10, 23, 1, 7, -4]



1. **Construcción del Max Heap:**
   A partir de la lista desordenada, se organiza en forma de Max Heap utilizando un proceso llamado heapify. El heapify toma un nodo, lo compara con sus hijos, y si alguno es mayor, lo intercambia y sigue bajando recursivamente hasta que se cumpla la propiedad de Max Heap.

2. **Extracción del elemento raíz:**
   Una vez formado el Max Heap, se intercambia la raíz (el mayor elemento) con el último elemento del heap. Luego se reduce el tamaño del heap (ignorando el último valor, que ya está ordenado) y se aplica nuevamente heapify para mantener la estructura.

3. **Repetición:**
   Este proceso se repite hasta que el heap se vacía y todos los elementos están ordenados de menor a mayor.

Ventajas:

- Eficiente y confiable, con complejidad garantizada O(n log n).
- In-place: no requiere memoria extra significativa.
- Útil en entornos con limitaciones de memoria.

Desventajas:

- No es estable, por lo que puede cambiar el orden relativo de elementos iguales.
- En la práctica, suele ser más lento que otros algoritmos con misma complejidad promedio, como Quick Sort o Merge Sort.

Complejidad:

- Peor caso: O(n log n)
- Mejor caso: O(n log n)
- Promedio: O(n log n)

Algoritmo	Complejidad (Promedio)	Estable	Uso de Memoria	Ventaja Principal	Desventaja Principal
Merge Sort	O(n log n)	Sí	Requiere memoria extra (O(n))	Rendimiento constante y estable	Mayor uso de memoria
Heap Sort	O(n log n)	No	In-place (sin memoria extra)	Eficiente sin consumir memoria adicional	No mantiene el orden de elementos iguales
Bubble Sort	$O(n^2)$	Sí	In-place (sin memoria extra)	Fácil de entender e implementar	Muy lento para listas grandes

¿Qué importancia tiene elegir el algoritmo de ordenamiento?

Elegir el algoritmo de ordenamiento adecuado es fundamental para garantizar la eficiencia y estabilidad de un sistema. La correcta selección impacta directamente en el tiempo de ejecución, el uso de recursos como memoria y procesamiento, y en la calidad del resultado, especialmente cuando se requiere mantener el orden relativo de los elementos iguales. Un algoritmo bien elegido optimiza el rendimiento, se adapta a las características de los datos y las restricciones del entorno, y asegura que el sistema funcione de manera rápida, confiable y escalable. Por lo tanto, comprender las ventajas y limitaciones de cada algoritmo es clave para tomar decisiones informadas que mejoren el desempeño global de cualquier aplicación o sistema complejo.

¿Cómo influye el contexto y las características de los datos en la elección del algoritmo de ordenamiento?

La elección del algoritmo de ordenamiento debe estar alineada con el contexto específico y las características de los datos a ordenar. Por ejemplo, si los datos ya están parcialmente ordenados, algoritmos adaptativos como Tim Sort pueden ofrecer mejoras significativas en rendimiento. En entornos con limitaciones de memoria, algoritmos in-place como Heap Sort son preferibles. Además, si es necesario mantener el orden original de elementos iguales, se debe optar por algoritmos estables como Merge Sort o Tim Sort. Por lo tanto, conocer las particularidades del conjunto de datos y las condiciones del sistema es esencial para seleccionar el algoritmo que maximice la eficiencia y la calidad del resultado.

Metodologías utilizadas

Durante la realización de este trabajo práctico integrador, se emplearon diferentes formas de organización y recursos que facilitaron tanto la investigación como la elaboración de los contenidos.

En primer lugar, se decidió dividir los temas entre los integrantes del grupo, con el objetivo de que cada uno pudiera enfocarse en un área específica. Esta elección permitió profundizar mejor en cada apartado y también hizo que el trabajo resultara más manejable para todos.

Por otro lado, se utilizaron herramientas de Inteligencia Artificial como apoyo para generar preguntas clave. Estas preguntas sirvieron como guía inicial para la búsqueda de información, ayudando a orientar la investigación de manera más clara y eficiente.

También se recurrió a material audiovisual, principalmente videos de YouTube, que ofrecieron explicaciones complementarias sobre los temas tratados. Estos contenidos ayudaron a comprender ciertos conceptos de forma más visual y accesible, reforzando lo trabajado a través de la lectura.

En conjunto, estas metodologías permitieron un desarrollo más ordenado y colaborativo del trabajo, combinando la distribución de tareas con el uso de recursos digitales para enriquecer el proceso.

Bibliografía

Geeks For Geeks
https://www.geeksforgeeks.org/linear-search/

Data Camp
https://www.datacamp.com/es/tutorial/linear-search-python

EDUCBA
https://www.educba.com/jump-search/

Study Tonight
https://www.studytonight.com/data-structures/jump-search-algorithm

Computer Science Bytes
https://www.computersciencebytes.com/sorting-algorithms/bubble-sort/

Michael Sambol
Bubble sort in 2 minutes

Merge sort in 3 minutes

Heap sort in 4 minutes

AlphaCodingSkills
https://www.alphacodingskills.com/php/pages/php-program-for-heap-sort.php