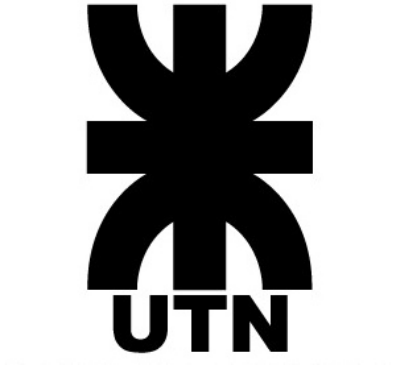
Algoritmos de Búsqueda y Ordenamiento



Alumnos

Nicolás Arrastía nicolasarrastia@gmail.com

Mariano Astorga marianoastorga46@gmail.com

Materia

Programación 1

Profesor

Bruselario Sebastián

Fecha de entrega

20 de junio de 2025

Índice

Introducción

En este trabajo integrador abordamos el estudio de los algoritmos de búsqueda y ordenamiento, una decisión que tomamos en conjunto como parte de nuestro proceso de formación en la carrera de Programación. Elegimos este tema porque creemos que estos algoritmos son fundamentales para el desarrollo de software eficiente, especialmente en lo que respecta al manejo y recuperación de datos.

A medida que avanzamos en la carrera, notamos que comprender cómo funcionan estos algoritmos no solo mejora el rendimiento de nuestras aplicaciones, sino que también nos prepara para trabajar con estructuras de datos más complejas, como árboles, grafos o bases de datos. Por eso, nos propusimos analizar los principales algoritmos de búsqueda y ordenamiento, evaluando sus ventajas, limitaciones y comportamiento.

Además del enfoque teórico, desarrollamos una aplicación práctica que nos permitió observar y comparar el funcionamiento de distintos algoritmos en un entorno controlado. A través de esta experiencia, no solo afianzamos los conocimientos adquiridos, sino que también pudimos comprobar en la práctica la eficacia de cada algoritmo en distintos contextos, evaluando su rendimiento según el tipo y la cantidad de datos. Esto nos ayudó a desarrollar una mirada más crítica sobre cuándo y por qué conviene utilizar cada uno.

Marco Teórico

Notación Big O

La notación Big O es una herramienta fundamental en el análisis de algoritmos, ya que permite expresar de forma precisa cómo crece el uso de recursos computacionales (principalmente tiempo de ejecución y memoria) a medida que aumenta el tamaño de los datos de entrada, representado como n. Más allá de los detalles específicos de una implementación o del

hardware utilizado, Big O proporciona una forma abstracta y estandarizada de comparar el rendimiento relativo de diferentes algoritmos.

Esta notación se centra en el **comportamiento asintótico** del algoritmo, es decir, en cómo se comporta cuando "n" tiende a crecer indefinidamente. Por eso, descarta factores constantes o términos de menor orden, ya que estos pierden relevancia frente al crecimiento de los datos. Por ejemplo, si un algoritmo tiene una complejidad de tiempo expresada como $3n^2 + 5n + 20$, su notación Big O será $O(n^2)$, porque es el término que domina el crecimiento.

Aunque tradicionalmente se asocia al peor caso (worst-case), Big O también puede utilizarse para describir el caso promedio o el mejor caso, según lo que se quiera analizar. Esta flexibilidad permite realizar análisis más realistas según la naturaleza del problema y el tipo de datos que se esperan en la práctica.

Complejidades más comunes

- O(1) – Tiempo constante

  El tiempo de ejecución no depende del tamaño de la entrada. Es el caso más eficiente. Por ejemplo, acceder a un elemento en un arreglo por índice.

- O(log n) – Tiempo logarítmico

  El tiempo crece muy lentamente en relación al tamaño de la entrada. Es característico de algoritmos como la búsqueda binaria, donde en cada paso se reduce a la mitad el espacio de búsqueda.

- O(n) – Tiempo lineal

  El tiempo de ejecución crece proporcionalmente con la entrada. Un ejemplo clásico es la búsqueda secuencial, donde se recorre cada elemento hasta encontrar el objetivo o agotar la lista.

- O(n log n) – Tiempo casi lineal

  Presente en algoritmos de ordenamiento eficientes como mergesort y heapsort. Es el límite inferior teórico para algoritmos de ordenamiento basados en comparaciones.

- $O(n^2)$ – Tiempo cuadrático

  El tiempo de ejecución crece rápidamente, ya que implica un número de operaciones proporcional al cuadrado del tamaño de la entrada. Aparece en algoritmos de fuerza bruta, como el ordenamiento por burbuja, que se vuelven inviables con grandes volúmenes de datos.

- $O(2^n)$, O(n!) – Tiempo exponencial y factorial

  Estas complejidades son altamente ineficientes y sólo aceptables en problemas pequeños o muy específicos. Se encuentran, por ejemplo, en algoritmos de fuerza bruta para resolver problemas de optimización combinatoria.

Aplicación práctica

En este trabajo, la notación Big O nos permitió evaluar de forma objetiva el desempeño de los algoritmos de búsqueda y ordenamiento implementados. Gracias a ella, fue posible anticipar el impacto del crecimiento de los datos sobre la eficiencia de cada algoritmo.

Comprender esta herramienta es clave en el desarrollo de software eficiente, ya que no se trata solo de que un algoritmo funcione, sino de que lo haga en un tiempo razonable según la escala del problema. Al comparar, por ejemplo, una búsqueda secuencial con una binaria, o un ordenamiento por burbuja (Bubble Sort) con uno por mezcla (Merge Sort), la notación Big O aporta un marco teórico que respalda las decisiones prácticas de implementación y optimización.

Big O no solo mide el rendimiento, también enseña a pensar en términos de escalabilidad, lo que resulta esencial tanto en proyectos simples como en sistemas complejos que manejan grandes volúmenes de información.

Algoritmos de Búsqueda

La búsqueda es una de las operaciones más esenciales en informática, ya que permite localizar un valor específico dentro de una estructura de datos, como listas, matrices, árboles o grafos. Esta operación se encuentra en el núcleo de múltiples procesos, desde la recuperación de información hasta la toma de decisiones en tiempo real.

En Python, es posible implementar diversos algoritmos de búsqueda, cada uno con características, ventajas y limitaciones particulares. Entre los más utilizados se encuentran:

Búsqueda Lineal (o Secuencial)

La búsqueda lineal es el método más básico y directo para localizar un elemento dentro de una colección de datos. Consiste en recorrer la estructura desde el primer elemento hasta el último, comparando cada uno con el valor buscado. La operación termina cuando se encuentra el elemento objetivo o cuando se ha revisado toda la lista sin éxito.

Funcionamiento:

Se inicia desde el índice cero y se compara cada elemento de forma secuencial con el valor buscado. Si se encuentra una coincidencia, se devuelve su posición (índice). Si no se encuentra en toda la estructura, se devuelve un valor especial como -1 para indicar que no está presente ya que no existe ese índice, a diferencia de 0 que sí existe, por eso la elección de -1.

Ventajas:

Una de las principales ventajas de la búsqueda lineal es su simplicidad. Puede aplicarse sobre cualquier tipo de arreglo, sin necesidad de que los datos estén ordenados previamente.

Además, su implementación es intuitiva y compatible con estructuras tanto simples como complejas.

Desventajas:

El mayor inconveniente radica en su ineficiencia con grandes volúmenes de datos, ya que su complejidad es lineal, lo que equivale a O(n) en la notación Big O. En el peor de los casos, es necesario examinar cada elemento, lo que genera un impacto negativo en el rendimiento si la búsqueda se repite con frecuencia o sobre estructuras extensas.

Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente diseñado para trabajar sobre estructuras de datos ordenadas de menor a mayor. Su lógica se basa en aplicar una estrategia de división sucesiva del espacio de búsqueda, lo que permite reducir drásticamente la cantidad de comparaciones necesarias para localizar un valor.

El proceso comienza identificando el elemento ubicado en el centro del arreglo. Si este coincide con el valor buscado, la búsqueda concluye. En caso contrario, se compara el valor objetivo con el elemento central: si es menor, se descarta la mitad derecha del arreglo; si es mayor, se descarta la mitad izquierda. Este procedimiento se repite en la mitad restante, de forma iterativa o recursiva, hasta encontrar el valor o agotar el espacio de búsqueda.

Si el elemento no está presente en la estructura, el algoritmo finaliza cuando los índices delimitadores se cruzan, lo que indica que ya no quedan elementos posibles por comparar. En ese caso, se concluye de manera eficiente que el valor no se encuentra en la colección. Esta capacidad de reducir el problema a la mitad en cada paso hace que la búsqueda binaria sea especialmente útil en estructuras grandes, alcanzando una complejidad logarítmica en notación Big O de O(log n).

Ventajas:

Su principal fortaleza es la eficiencia: gracias a su enfoque de reducción del espacio de búsqueda, la complejidad se reduce a O(log n), lo que permite obtener resultados mucho más rápidos en estructuras grandes. Además, es ideal para contextos donde se realizan búsquedas frecuentes sobre datos ordenados.

Desventajas:

A pesar de su eficiencia, la búsqueda binaria no puede utilizarse sobre datos desordenados, lo que implica una limitación importante. Si los datos no están organizados previamente, se requiere un proceso adicional de ordenamiento que puede impactar negativamente en el rendimiento general, sobre todo si solo se necesita una única búsqueda.

Comparación y contexto de uso

La elección entre búsqueda lineal y binaria depende principalmente de la estructura de datos y de si esta está ordenada. La búsqueda lineal es útil cuando los datos son pequeños o cuando el costo de ordenar no está justificado. En cambio, la búsqueda binaria es la opción preferida en grandes conjuntos ordenados por su eficiencia y velocidad.

Además, la búsqueda binaria forma la base de algoritmos y estructuras de datos más complejas, como árboles binarios de búsqueda y tablas hash ordenadas, mientras que la búsqueda lineal sigue siendo útil para escenarios simples o cuando la ordenación no es posible o práctica.

Aplicaciones comunes de los algoritmos de búsqueda:

Las aplicaciones comunes de los algoritmos de búsqueda incluyen la localización de palabras clave en textos o documentos, la búsqueda de archivos dentro de un sistema operativo, la consulta de registros específicos en una base de datos, la determinación de rutas óptimas en grafos, como ocurre en sistemas de navegación GPS como Google Maps, y la resolución de problemas de optimización mediante la búsqueda de soluciones viables.

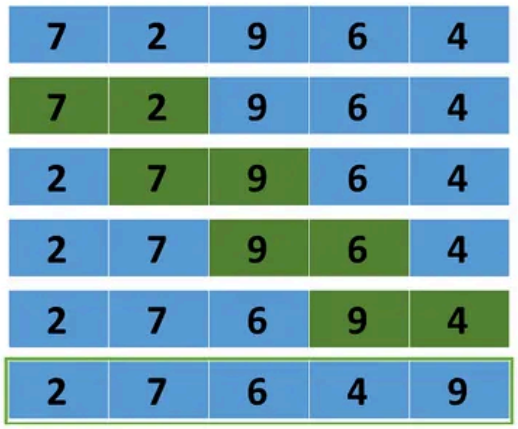
Entender los distintos algoritmos de búsqueda permite seleccionar el más adecuado según la estructura de datos y el contexto del problema. Al comprender cómo funcionan internamente y qué impacto tienen en el rendimiento, es posible escribir programas más eficientes, robustos y escalables.

Algoritmos de Ordenamiento

El ordenamiento de datos es una operación fundamental en programación. Permite reorganizar colecciones según un criterio específico (generalmente ascendente o descendente), lo cual no solo facilita su interpretación y procesamiento, sino que también habilita el uso de algoritmos más eficientes, como la búsqueda binaria. Su aplicación es clave en múltiples áreas, desde sistemas de bases de datos hasta inteligencia artificial, y su elección impacta directamente en el rendimiento de cualquier sistema que manipule grandes volúmenes de información.

A continuación, se presentan algunos de los algoritmos de ordenamiento más conocidos y utilizados:

Bubble Sort (Ordenamiento de burbuja)



Selection Sort es un algoritmo de ordenamiento que funciona seleccionando, en cada paso, el elemento más pequeño (o el más grande, dependiendo del criterio) de la porción desordenada de la lista y colocándolo en su posición definitiva dentro de la porción ordenada. Para lograr esto, divide conceptualmente el arreglo en dos segmentos: una sección inicial ordenada, que se va construyendo de izquierda a derecha, y una sección desordenada que contiene los elementos restantes.

El algoritmo comienza localizando el valor mínimo de toda la lista. Una vez encontrado, lo intercambia con el primer elemento. En la siguiente iteración, busca el menor dentro del resto de la lista, lo coloca en la segunda posición, y así sucesivamente. El proceso continúa hasta que todos los elementos se han colocado en el lugar correspondiente, logrando así una lista completamente ordenada.

La complejidad temporal de Selection Sort es siempre $O(n^2)$, ya que, sin importar el orden inicial de los datos, realiza la misma cantidad de comparaciones. Este comportamiento predecible lo hace útil en contextos donde se requiere simplicidad y no es necesario optimizar la velocidad para grandes volúmenes.

Ventajas:

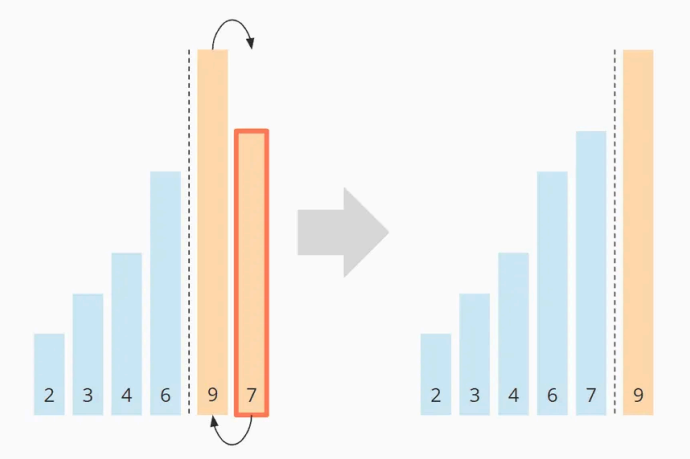
Su principal fortaleza reside en su simplicidad. Es fácil de entender, implementar y enseñar, lo que lo convierte en una opción común en etapas iniciales del aprendizaje de algoritmos o para arreglos muy cortos.

Desventajas:

No es eficiente para listas largas o con datos muy desordenados, ya que su complejidad temporal es $O(n^2)$ en el peor y promedio de los casos. Además, realiza muchos intercambios innecesarios, lo que afecta negativamente el rendimiento en comparación con otros algoritmos más avanzados.

Selection Sort (Ordenamiento por selección)

Selection Sort es un algoritmo de ordenamiento que funciona seleccionando, en cada paso, el elemento más pequeño (o el más grande, dependiendo del criterio) de la porción desordenada de la lista y



colocándolo en su posición definitiva dentro de la porción ordenada. Para lograr esto, divide conceptualmente el arreglo en dos segmentos: una sección inicial ordenada, que se va construyendo de izquierda a derecha, y una sección desordenada que contiene los elementos restantes.

El algoritmo comienza localizando el valor mínimo de toda la lista. Una vez encontrado, lo intercambia con el primer elemento. En la siguiente iteración, busca el menor dentro del resto de la lista, lo coloca en la segunda posición, y así sucesivamente. El proceso continúa hasta que todos los elementos se han colocado en el lugar correspondiente, logrando así una lista completamente ordenada.

La complejidad temporal de Selection Sort es siempre $O(n^2)$, ya que, sin importar el orden inicial de los datos, realiza la misma cantidad de comparaciones. Este comportamiento predecible lo hace útil en contextos donde se requiere simplicidad y no es necesario optimizar la velocidad para grandes volúmenes.

Ventajas:

Una de las principales ventajas de Selection Sort es su facilidad de implementación y su comportamiento determinista. Al realizar un número fijo de comparaciones, su ejecución es consistente. Además, realiza un número reducido de intercambios en comparación con otros algoritmos básicos, lo que puede ser útil en contextos donde los costos de escritura en memoria son significativos.

Desventajas:

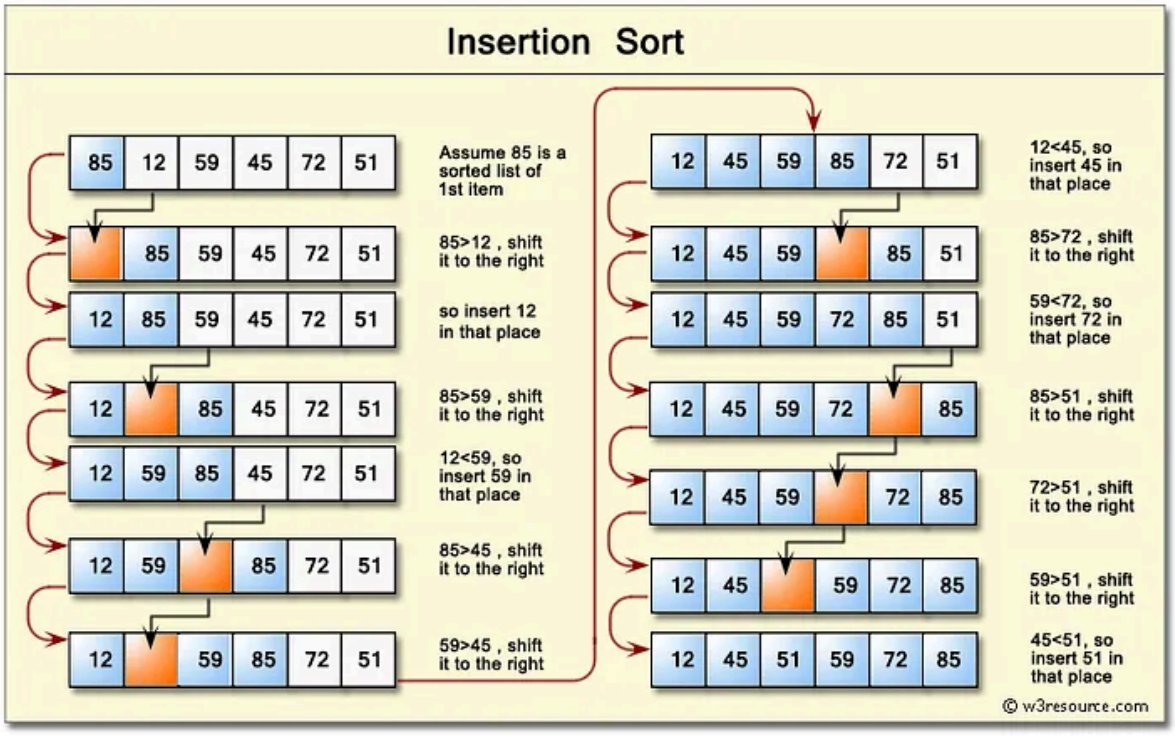
Sin embargo, su mayor desventaja radica en su ineficiencia con listas grandes. El número de comparaciones crece cuadráticamente con el tamaño de la lista, lo que lo vuelve poco práctico cuando se trabaja con grandes volúmenes de datos. Además, a diferencia de otros métodos como Insertion Sort, no es un algoritmo estable: no garantiza que los elementos iguales conserven su orden relativo.

Insertion Sort (Ordenamiento por inserción)

Insertion Sort es un algoritmo de ordenamiento simple pero efectivo en ciertos contextos. Su funcionamiento se basa en construir progresivamente una lista ordenada, tomando un elemento a la vez y ubicándolo en la posición correcta respecto a los elementos anteriores. El proceso inicia asumiendo que el primer elemento ya está ordenado. Luego, el algoritmo toma el siguiente elemento y lo compara con los de su izquierda, desplazando hacia la derecha aquellos que sean mayores, hasta encontrar el lugar exacto donde insertarlo. Este proceso se repite con cada nuevo elemento, expandiendo gradualmente la porción ordenada de la lista.

La lógica detrás de Insertion Sort es similar a cómo una persona ordena manualmente un mazo de cartas: mantiene una parte ordenada y va insertando cada nueva carta en el lugar que le corresponde. Este enfoque lo hace intuitivo y fácil de implementar, especialmente para quienes están comenzando en programación.

La complejidad temporal en el peor caso es $O(n^2)$, ya que cada elemento puede llegar a compararse con todos los anteriores. Sin embargo, en listas pequeñas o que ya están casi ordenadas, su desempeño mejora considerablemente, alcanzando tiempos cercanos a $O(n)$ en el mejor escenario.



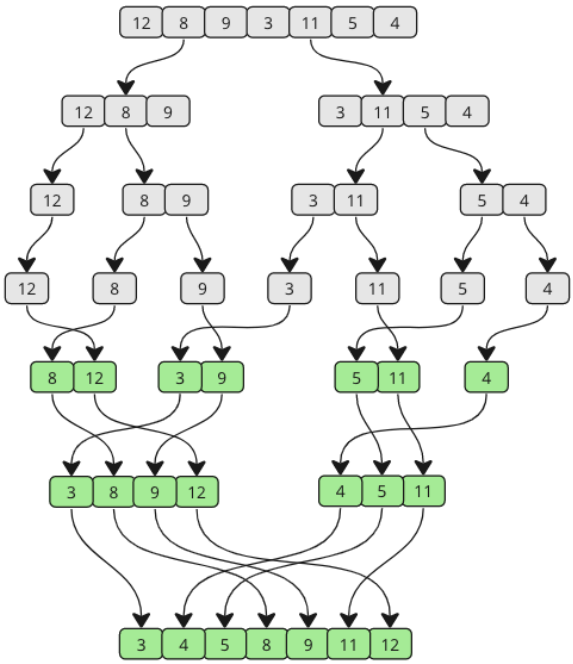
Ventajas:

Insertion Sort no requiere memoria adicional, ya que realiza el ordenamiento *in-place*, lo que lo convierte en una buena opción cuando se dispone de pocos recursos. Además, es un algoritmo estable, es decir, conserva el orden relativo de elementos iguales. Otra ventaja importante es su excelente comportamiento en listas cortas o casi ordenadas, donde puede superar en eficiencia a algoritmos más complejos.

Desventajas:

El mayor inconveniente de Insertion Sort aparece con listas largas y desordenadas, donde su rendimiento se degrada notablemente debido al elevado número de comparaciones y desplazamientos necesarios. Su complejidad cuadrática lo hace ineficiente frente a algoritmos como Merge Sort o Quick Sort en escenarios de gran volumen de datos.

Merge Sort (Ordenamiento por mezcla)



Merge Sort es un algoritmo de ordenamiento estable y eficiente que se caracteriza por su enfoque sistemático y predecible. Su funcionamiento se basa en descomponer el problema en partes cada vez más pequeñas: comienza dividiendo la lista original en mitades de forma recursiva, hasta obtener listas mínimas formadas por un solo elemento. Estas pequeñas listas, al contener solo un valor, se consideran naturalmente ordenadas.

Una vez alcanzado ese punto, el algoritmo inicia la etapa clave: el proceso de mezcla. Durante esta fase, se toman pares de listas ordenadas y se fusionan en una nueva lista, también ordenada, comparando elemento por elemento. Esta operación se repite hasta reconstruir la lista original, pero ahora completamente ordenada. A diferencia de otros métodos que reorganizan directamente los datos, Merge Sort realiza una copia temporal de los elementos durante la combinación, lo que asegura que el orden final sea correcto.

Su complejidad temporal es O(n log n) en todos los casos, incluso en su peor escenario, lo que lo convierte en una opción muy confiable para datos de gran tamaño o para estructuras que requieren ordenamientos estables (como bases de datos).

Ventajas

Una de sus principales fortalezas es la consistencia en su rendimiento. A diferencia de algoritmos como Quick Sort, su tiempo de ejecución no depende de la distribución inicial de los datos. Además, es estable: conserva el orden relativo de los elementos iguales, algo fundamental en ciertos contextos donde cada elemento lleva más información asociada.

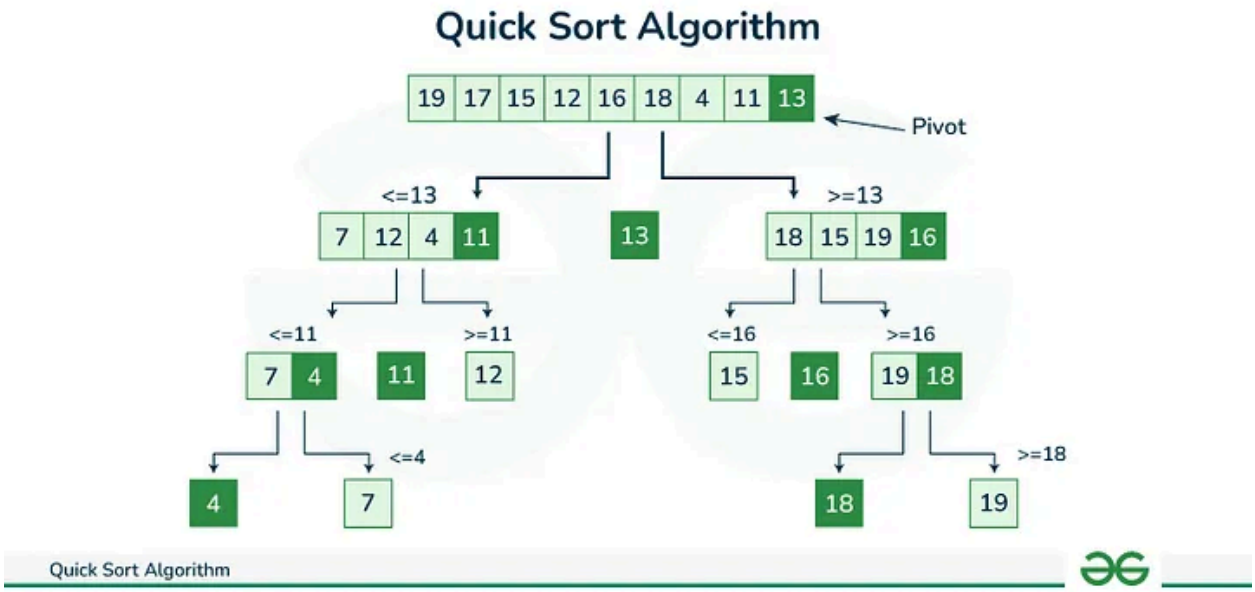
Desventajas

El principal inconveniente de Merge Sort es su uso de memoria adicional. Para realizar las fusiones de forma ordenada, requiere espacio auxiliar proporcional al tamaño de la lista original. Esto lo vuelve menos ideal en sistemas con memoria limitada o cuando se trabaja con estructuras muy grandes en entornos con restricciones de recursos. También puede resultar menos eficiente que Quick Sort en listas pequeñas, debido al costo que implica crear estructuras temporales.

Quick Sort (Ordenamiento rápido)

Quick Sort es un algoritmo de ordenamiento muy eficiente que organiza los elementos de una lista en función de un valor denominado pivote. A partir de ese valor, redistribuye los elementos: todos los menores al pivote se ubican a su izquierda, y los mayores, a su derecha. Este procedimiento de partición se repite de forma independiente sobre cada una de las dos mitades, hasta que toda la estructura queda ordenada. A diferencia de otros algoritmos que requieren espacio adicional, Quick Sort modifica el arreglo directamente, sin necesidad de estructuras auxiliares importantes.

El rendimiento del algoritmo está estrechamente ligado a la elección del pivote. Si se logra dividir la lista en partes de tamaño similar, el algoritmo alcanza una eficiencia promedio de O(n log n). Sin embargo, una mala elección (como tomar siempre el primer o último elemento en listas ya ordenadas) puede producir divisiones muy desbalanceadas y elevar la complejidad hasta $O(n^2)$. Para evitar esto, suelen utilizarse técnicas como el pivote aleatorio o la mediana de tres.



Ventajas:

Una de las principales ventajas de Quick Sort es su velocidad en escenarios promedio y su bajo consumo de memoria, ya que no requiere crear nuevas listas durante el ordenamiento. Su desempeño lo convierte en una excelente elección para grandes volúmenes de datos y es, por eso, muy utilizado en implementaciones reales.

Desventajas:

Entre sus desventajas, se destaca el hecho de que su rendimiento se puede degradar considerablemente si la elección del pivote no es adecuada. Además, su implementación recursiva puede generar un uso elevado de la pila de llamadas, lo cual puede resultar problemático en sistemas con recursos limitados si no se toman precauciones como establecer un límite de recursión o usar una versión iterativa.

Importancia del Ordenamiento

El ordenamiento de datos es una operación fundamental en informática, y su correcta implementación tiene un impacto directo en el rendimiento global de los sistemas. Dominar los

algoritmos de ordenamiento permite a los programadores optimizar tareas como la búsqueda, el análisis y la visualización de información. Un conjunto de datos bien ordenado facilita tanto su interpretación por parte de los usuarios como su procesamiento por parte del sistema.

Además de acelerar los tiempos de ejecución y reducir el consumo de memoria, el ordenamiento eficiente permite escalar soluciones a contextos más complejos, como grandes bases de datos, procesamiento en tiempo real o sistemas con recursos limitados. En campos como la inteligencia artificial, la minería de datos y los motores de búsqueda, el ordenamiento actúa como una etapa previa indispensable para tareas más avanzadas.

Elegir el algoritmo adecuado según el tipo, volumen y distribución de los datos, así como según las restricciones del entorno, es una habilidad clave para cualquier programador que aspire a desarrollar software robusto, rápido y escalable.

Comparación y criterios de selección de algoritmos

No existe un algoritmo de ordenamiento universalmente superior; cada uno presenta ventajas y desventajas según el caso de uso. Por eso, es importante saber evaluar factores como la complejidad temporal, el consumo de memoria, la estabilidad (si mantiene el orden relativo entre elementos iguales) y si opera in-place (sin estructuras auxiliares).

Por ejemplo, en listas pequeñas o casi ordenadas, Insertion Sort puede superar a algoritmos más complejos. Para grandes volúmenes de datos, Merge Sort y Quick Sort suelen ser preferidos, aunque el primero requiere memoria adicional y el segundo depende críticamente de la elección del pivote. En sistemas con recursos limitados, algoritmos simples como Selection Sort podrían ser suficientes.

Comprender estos criterios permite tomar decisiones fundamentadas y construir soluciones eficientes adaptadas al problema real.

Caso Práctico

Con el objetivo de observar y analizar el comportamiento real de distintos algoritmos de búsqueda y ordenamiento, desarrollamos un programa en Python que permitió poner en práctica los conceptos estudiados en el marco teórico.

Generación de datos

El primer paso consistió en generar una lista aleatoria de números enteros sin repeticiones, utilizando una permutación de valores consecutivos del 1 al n. Para este caso, se definió n = 2000, lo que garantizó un conjunto acotado y controlado de datos. Además, se seleccionó aleatoriamente un valor dentro de esa lista para utilizarlo como objetivo en las búsquedas posteriores.

Ejecución de los algoritmos

Se implementaron cinco algoritmos de ordenamiento: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort y Quick Sort. Cada uno fue ejecutado sobre la misma lista desordenada, lo que permitió evaluar sus diferencias en rendimiento y comportamiento práctico. Luego de cada ordenamiento, se aplicaron dos algoritmos de búsqueda: Lineal y Binaria. En el caso de la búsqueda binaria, se utilizó la versión ya ordenada de la lista, requisito fundamental para su correcto funcionamiento.

Medición de resultados

Durante la ejecución, se midió el tiempo de ejecución de cada algoritmo utilizando la biblioteca time de Python. Además, se instrumentaron temporalmente los algoritmos para contar la cantidad de comparaciones realizadas, lo cual nos permitió observar de forma más tangible cómo varía el costo computacional entre ellos. Aunque esta métrica no se mantendrá en la versión final del código, resultó útil para fines comparativos en esta instancia.

Visualización

Para representar visualmente los resultados obtenidos, se empleó la biblioteca matplotlib para generar gráficos de barras comparativas, en los que se contrastaron los tiempos de ejecución y la cantidad de comparaciones de cada algoritmo. Esta visualización facilitó la interpretación de los datos y reforzó los conceptos teóricos a través de evidencia empírica.

Evaluación de resultados

Los resultados obtenidos permitieron validar en la práctica lo aprendido en la teoría. Se comprobó que los algoritmos con mayor complejidad en notación Big O (como Bubble Sort o Selection Sort) requerían más tiempo y comparaciones, especialmente en ejecuciones repetidas con listas de mayor tamaño. Por el contrario, algoritmos más eficientes como Merge Sort y Quick Sort demostraron un rendimiento superior en escenarios con muchos datos. En cuanto a las búsquedas, la binaria fue notablemente más eficiente que la lineal cuando se trabajaba sobre listas ordenadas, tal como se esperaba.

Posibles mejoras y extensiones del caso práctico

Si bien el caso práctico actual permitió validar de manera efectiva el comportamiento teórico de los algoritmos implementados, existen múltiples formas de enriquecer y profundizar el análisis. Una mejora significativa sería incorporar casos en los que el valor buscado no se encuentra en la lista, lo que permitiría observar el rendimiento de los algoritmos de búsqueda en su peor escenario. Esto es especialmente relevante para la búsqueda lineal, que en dicho caso debe recorrer toda la lista, y para la binaria, que agota el rango hasta determinar la ausencia del elemento.

Otra extensión interesante sería aumentar el tamaño de los datos de entrada. Probar los algoritmos con listas de cientos o miles de elementos permitiría visualizar con mayor claridad cómo se manifiesta la eficiencia (o ineficiencia) de cada uno, especialmente en lo que respecta a la complejidad temporal expresada en notación Big O.

También se podrían evaluar distintos patrones de ordenación inicial, como listas ya ordenadas, casi ordenadas o completamente invertidas, para observar cómo estas condiciones afectan el comportamiento de cada algoritmo, en particular de Insertion Sort y Quick Sort, que son sensibles a la disposición inicial de los datos.

Por último, se podría incorporar una métrica adicional como el uso de memoria o realizar una comparación entre versiones iterativas y recursivas de ciertos algoritmos, para ampliar aún más la evaluación del desempeño general. Estas variantes permitirían obtener una perspectiva más completa y realista del impacto de las decisiones algorítmicas en el desarrollo de software.

Metodología Utilizada

Para realizar este trabajo seguimos una serie de pasos que combinan teoría y práctica. Investigamos los algoritmos más conocidos, los implementamos en Python, probamos su rendimiento con distintos datos y registramos los resultados en gráficos. Todo esto nos permitió entender mejor cómo funcionan en la práctica.

Recopilación Teórica

La base del proyecto comenzó con una investigación exhaustiva sobre algoritmos de ordenamiento y búsqueda. Se partió del material del temario, que incluye los algoritmos más comunes y relevantes para entender los fundamentos teóricos y prácticos del tema. Para complementar, se recurrió a libros especializados, artículos científicos y documentación técnica confiable disponible en línea, lo que permitió tener una visión clara y detallada de cómo funciona cada algoritmo, su complejidad temporal y espacial, así como sus ventajas y limitaciones. Este análisis previo fue fundamental para seleccionar qué algoritmos implementar y cómo evaluarlos correctamente, asegurando que el desarrollo práctico esté sustentado en conocimiento sólido.

Desarrollo en Python

La implementación de los algoritmos se realizó íntegramente en Python, un lenguaje elegido por su simplicidad, legibilidad y la facilidad para manejar estructuras de datos. El entorno de desarrollo principal fue Visual Studio Code, que facilitó la escritura, depuración y organización del código. Se diseñaron funciones modulares para cada algoritmo, lo que permitió aislar la lógica y hacer pruebas individuales sin interferencias. Para medir la eficiencia, se creó una función timer() que ejecuta un algoritmo, mide el tiempo transcurrido y devuelve ambos resultados, permitiendo así la comparación directa entre ellos sin alterar el comportamiento original del código. Además, se agregaron contadores de operaciones (comparaciones) para profundizar el análisis, aunque eventualmente se decidió dejar solo la medición temporal para evitar sobrecargar las pruebas.

Ejecución y Pruebas

Para evaluar el desempeño de cada algoritmo, se generaron listas con 2000 números enteros únicos, asegurando que los datos cubrieran un rango suficiente para observar diferencias de comportamiento entre algoritmos. Cada algoritmo se ejecutó múltiples veces fuera de la grabación para obtener un promedio estadísticamente más fiable, minimizando así el impacto de posibles fluctuaciones en el sistema. La búsqueda binaria se aplicó exclusivamente sobre listas previamente ordenadas, dado a que es un requerimiento. Mientras que la búsqueda lineal se probó con listas ordenadas principalmente, para compararla con la búsqueda binaria, pero también se probó con listas desordenadas. También se exploraron escenarios donde el valor buscado no estaba presente, aunque estos datos no se presentaron en el video. Este enfoque permitió verificar la validez de la teoría de la notación Big O en la práctica y entender mejor cómo varía el rendimiento con el tamaño y orden de los datos.

Visualización de Resultados

Para comunicar los resultados de manera clara y visual, se empleó la biblioteca matplotlib de Python, que permitió crear gráficos de barras comparativos mostrando el tiempo promedio de ejecución de cada algoritmo. Estos gráficos sirvieron para validar empíricamente las expectativas teóricas: algoritmos con complejidad $O(n^2)$ mostraron un desempeño significativamente peor frente a otros con complejidad $O(n \log n)$ para el mismo conjunto de datos. La visualización ayudó a interpretar mejor las diferencias de eficiencia y a demostrar que la selección adecuada del algoritmo depende del contexto y el tamaño de los datos, reforzando así el valor práctico del estudio.

Producción de la Presentación

Como parte integral del proyecto, se preparó una presentación en Google Slides para organizar y resumir la información de forma estructurada y visualmente atractiva, facilitando la comunicación de los conceptos, metodología y resultados a una audiencia externa. Además, se produjo un video explicativo donde se mostraron en acción los algoritmos, brindando un soporte visual que complementa la teoría y los datos. Aunque el video no aportó datos adicionales, cumplió una función didáctica importante, permitiendo observar la ejecución paso a paso y ayudando a clarificar el funcionamiento de los algoritmos para quienes no estén familiarizados con el código.

Resultados Obtenidos

Análisis de Resultados

Luego de implementar y evaluar diversos algoritmos de búsqueda y ordenamiento, se obtuvieron resultados concretos que permiten comparar su desempeño en términos de eficiencia temporal y cantidad de comparaciones realizadas. Las pruebas se realizaron con

múltiples ejecuciones para garantizar la consistencia de los datos y reducir el impacto de posibles variaciones aleatorias.

Algoritmos de Ordenamiento

Se analizaron cinco algoritmos clásicos: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort y Quick Sort. Los experimentos se realizaron sobre listas de diferente tamaño y nivel de desorden para observar cómo se comporta cada algoritmo bajo condiciones variables.

Bubble Sort resultó ser el menos eficiente del conjunto, tanto en tiempo de ejecución como en cantidad de comparaciones. Esto se debe a su naturaleza cuadrática ($O(n^2)$) y a su necesidad de recorrer repetidamente la lista, intercambiando elementos adyacentes incluso cuando la lista ya se encuentra parcialmente ordenada. Su uso práctico se limita a contextos educativos o estructuras muy pequeñas.

Insertion Sort mostró un rendimiento ligeramente superior a Bubble Sort, especialmente en listas parcialmente ordenadas, donde logra reducir el número de operaciones. Sin embargo, mantiene una complejidad cuadrática en el peor de los casos. Su simpleza lo vuelve útil cuando se trabaja con volúmenes reducidos de datos o cuando la lista ya está casi ordenada.

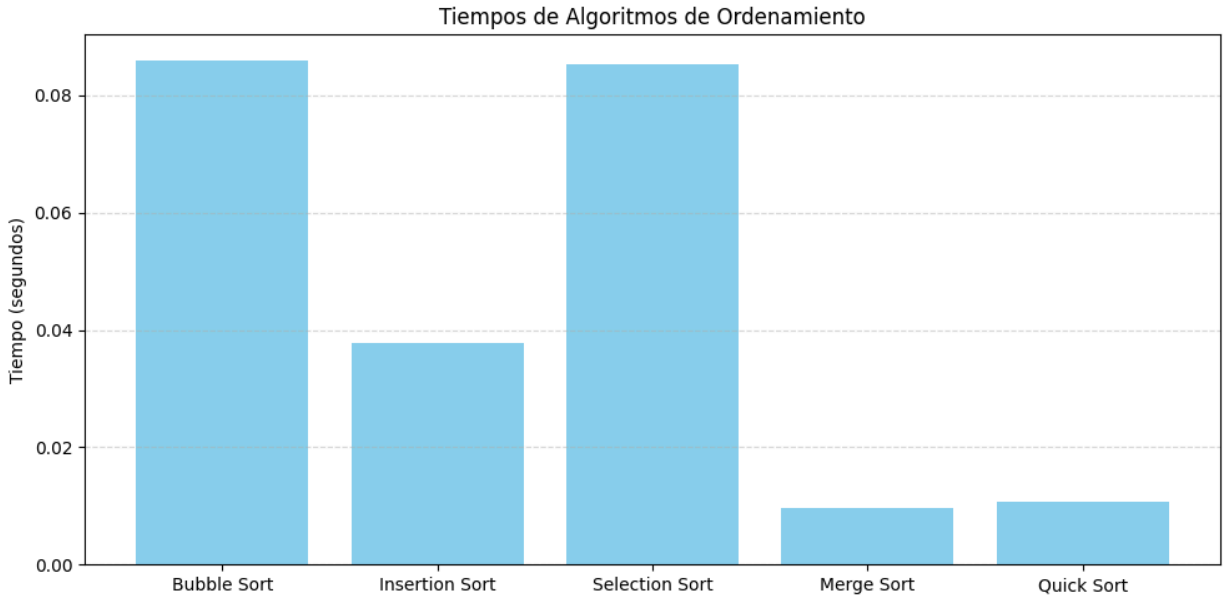
Selection Sort, a pesar de realizar un número constante de comparaciones independientemente del orden inicial, no superó a Insertion Sort en tiempo total de ejecución. Esto se debe a que sus intercambios implican recorrer la lista completa para encontrar el mínimo, lo que también incurre en una complejidad de $O(n^2)$.

Merge Sort se destacó por su eficiencia y consistencia. Su enfoque de dividir y combinar logró mantener bajos tiempos de ejecución incluso con listas grandes. Gracias a su complejidad $O(n \log n)$, demostró ser ideal para estructuras extensas donde el ordenamiento rápido y estable es prioritario.

Quick Sort, en sus mejores escenarios, rivalizó con Merge Sort e incluso lo superó en velocidad, especialmente con listas medianas. Sin embargo, en casos desfavorables (como listas ya

ordenadas sin una estrategia de pivote adecuada), mostró caídas de rendimiento. Aun así, su uso es muy común debido a su rapidez promedio y su baja utilización de memoria adicional.

Los resultados obtenidos pueden visualizarse con mayor claridad en el gráfico comparativo, donde se destacan las diferencias entre algoritmos según el tamaño de la lista y la cantidad de operaciones realizadas.



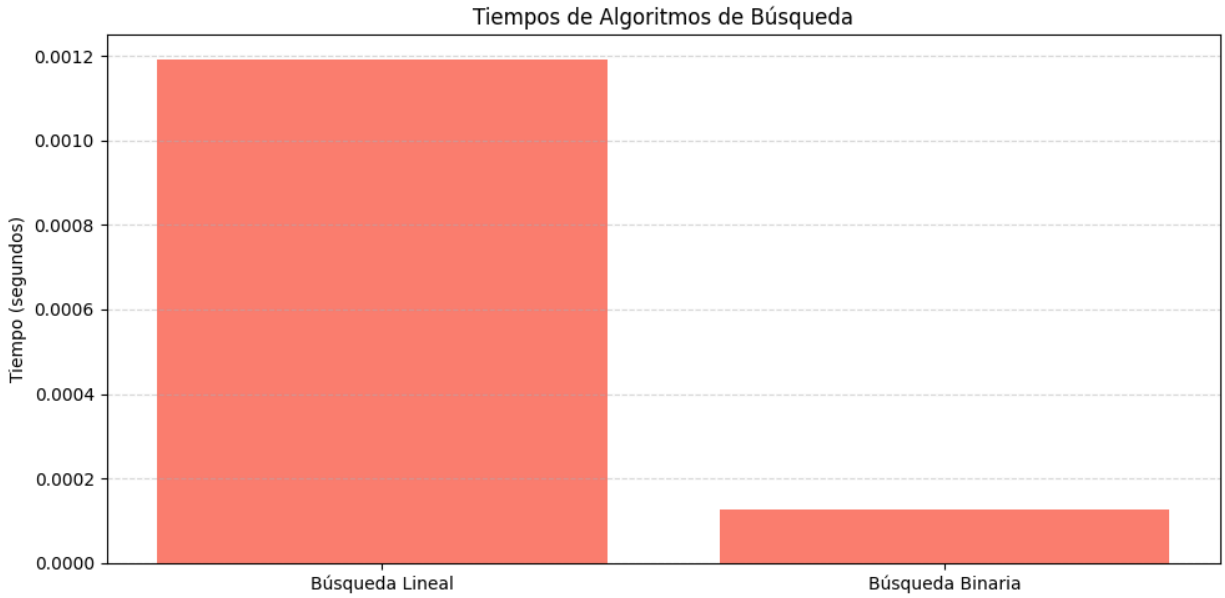
Algoritmos de Búsqueda

En cuanto a las búsquedas, se evaluaron dos estrategias fundamentales: búsqueda lineal y búsqueda binaria, aplicadas sobre listas de distintos tamaños y grados de orden.

La búsqueda lineal mostró un rendimiento aceptable en estructuras pequeñas, pero su eficiencia se degradó rápidamente conforme aumentó el número de elementos. Esto se debe a su naturaleza secuencial, que implica recorrer todos los elementos hasta hallar el valor buscado o agotar la lista. Su complejidad O(n) la vuelve poco recomendable para estructuras grandes sin ordenar.

La búsqueda binaria, en contraste, ofreció resultados significativamente superiores en cuanto a velocidad y cantidad de comparaciones. Al operar mediante una estrategia de reducción progresiva del espacio de búsqueda, logró identificar elementos rápidamente, siempre que la lista estuviera ordenada previamente. Su complejidad O(log n) se mantuvo constante, incluso en listas extensas.

Los gráficos de esta sección reflejan claramente cómo la búsqueda binaria supera ampliamente a la búsqueda lineal en contextos donde se cumple la condición del orden, consolidándose como una herramienta fundamental en operaciones sobre grandes volúmenes de datos.



Conclusiones

Los resultados empíricos confirman las predicciones teóricas: los algoritmos cuadráticos como Bubble, Insertion y Selection presentan limitaciones severas en términos de escalabilidad, mientras que estrategias más avanzadas como Merge Sort y Quick Sort ofrecen mejoras sustanciales en eficiencia. Del mismo modo, la búsqueda binaria se consolida como una solución de alta eficiencia siempre que se trabaje sobre estructuras ordenadas, en contraste con la búsqueda lineal, que queda relegada a contextos simples o sin orden previo.

Estas pruebas no solo validan el comportamiento esperado de cada algoritmo, sino que también aportan una visión práctica sobre cuál utilizar según el contexto y las características del problema a resolver.

Conclusiones Finales

El desarrollo de este proyecto permitió explorar en profundidad el comportamiento, la eficiencia y la aplicabilidad de diversos algoritmos de búsqueda y ordenamiento, implementados en el lenguaje de programación Python. A partir de las pruebas realizadas y el análisis de resultados, se obtuvieron conclusiones relevantes que refuerzan tanto los conceptos teóricos como su puesta en práctica en contextos reales.

Comparación de rendimiento y eficiencia

Los algoritmos evaluados demostraron diferencias notables en cuanto a rendimiento, especialmente al variar el tamaño y la naturaleza de los conjuntos de datos. En el caso de la búsqueda, la búsqueda binaria se mostró ampliamente superior a la búsqueda lineal en términos de tiempo de respuesta, siempre que los datos estuvieran previamente ordenados. Esta condición, aunque puede parecer una limitación, también resalta la importancia de un ordenamiento previo eficiente.

En cuanto a los algoritmos de ordenamiento, se confirmó que métodos como Bubble Sort, Insertion Sort y Selection Sort, si bien son fáciles de implementar y comprender, presentan un rendimiento deficiente en listas grandes debido a su complejidad cuadrática. Por el contrario, Merge Sort y Quick Sort ofrecieron tiempos considerablemente más bajos en escenarios de gran volumen, gracias a su complejidad O(n log n). No obstante, también se evidenció que Quick Sort puede sufrir degradación de rendimiento si no se elige bien el pivote, especialmente en listas ya ordenadas o casi ordenadas, donde Merge Sort se comporta de forma más estable.

Implementación en Python

Python resultó ser una herramienta adecuada para la implementación de estos algoritmos. Su sintaxis clara y expresiva, junto a la disponibilidad de bibliotecas estándar como “time” y “random”, permitió desarrollar código funcional y medir de manera precisa los tiempos de ejecución. Además, la facilidad para trabajar con estructuras como listas y funciones recursivas agilizó el desarrollo y facilitó el enfoque modular del código. Este entorno de trabajo favoreció la experimentación con diferentes técnicas y variantes sin complejidad adicional.

Validación teórica a través de la práctica

La experiencia adquirida a lo largo del proyecto permitió comprobar empíricamente las complejidades asociadas a cada algoritmo, reforzando la teoría estudiada. En todos los casos, los resultados obtenidos fueron coherentes con el análisis de complejidad Big O previsto. Esta validación práctica no sólo afianzó el conocimiento sobre cómo funcionan los algoritmos, sino también sobre cuándo y por qué conviene utilizar cada uno según el contexto específico.

Aplicación en escenarios reales

Los algoritmos analizados no solo representan contenido fundamental en la formación académica, sino que tienen aplicaciones directas en la industria del software. Desde sistemas de búsqueda en bases de datos hasta algoritmos de recomendación o rutas óptimas en sistemas de navegación, la correcta elección e implementación de estos métodos influye directamente en la eficiencia, escalabilidad y calidad de las soluciones informáticas desarrolladas.

Proyecciones y mejoras posibles

Si bien los objetivos propuestos se cumplieron satisfactoriamente, existen oportunidades de mejora y ampliación. Una línea futura interesante sería la incorporación de algoritmos híbridos (como Timsort) o paralelos, capaces de aprovechar múltiples núcleos de procesamiento en escenarios de grandes volúmenes de datos. También sería valioso optimizar el uso de memoria,

especialmente en algoritmos recursivos como Quick Sort y Merge Sort, y profundizar en estrategias de ordenamiento adaptativo que puedan responder dinámicamente a la estructura de los datos de entrada.

En síntesis, este proyecto permitió no solo consolidar conocimientos teóricos, sino también adquirir una perspectiva crítica sobre el impacto que una decisión algorítmica puede tener en el desarrollo de software eficiente, robusto y escalable.

Bibliografía

Python Software Foundation. (2025, Junio 5). *Python 3 Library Documentation*. https://docs.python.org/3/library/

UdiProd. (2025, Junio 6). *Bubble Sort Algorithm Explained* [Video]. YouTube. https://www.youtube.com/watch?v=TZRWRjq2CAg

Bro Code. (2025, Junio 6). *Selection Sort Algorithm Explained* [Video]. YouTube. https://www.youtube.com/watch?v=Vtckgz38QHs

Bro Code. (2025, Junio 6). *Insertion Sort Algorithm Explained* [Video]. YouTube. https://www.youtube.com/watch?v=3j0SWDX4AtU

Sonu Me. (2025, Junio 16). *Understanding Bubble Sort Algorithm: A Step-by-Step Guide*. Medium. https://medium.com/@me.sonu300/understanding-bubble-sort-algorithm-a-step-by-step-guide-12888dc84d13

Lee, Y. (2025, Junio 16). *Selection Sort Algorithm*. Medium. https://yuminlee2.medium.com/selection-sort-algorithm-a4b59fd2ba35

Basu Binayak. (2025, Junio 16). *Sorting: Insertion Sort*. Medium. https://medium.com/@basubinayak05/sorting-insertion-sort-870fe06163e3

W3Schools. (2025, Junio 16). *Merge Sort Algorithm*.

https://www.w3schools.com/dsa/dsa_algo_mergesort.php

Wikipedia. (2025, Junio 17). *Big O notation*.

https://en.wikipedia.org/wiki/Big_O_notation

Anexo

Repositorio del código en GitHub

https://github.com/NicolasArrastia/UTN-Prog1-Integrador

Video explicativo del proyecto

https://youtu.be/AtE0HO01RNg

Presentación en diapositivas

https://docs.google.com/presentation/d/1vw2DWgMxB9wmQVKOFWWiCv0fjh2i19mdQiGEWKv2NLc/edit?usp=drive_link