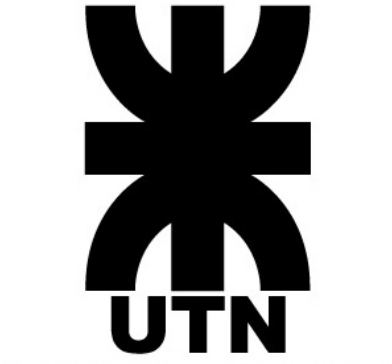
Algoritmos de Búsqueda y Ordenamiento



Alumnos

Nicolás Arrastía nicolasarrastia@gmail.com

Mariano Astorga marianoastorga46@gmail.com

Materia

Programación 1

Profesor

Bruselario Sebastián

Fecha de entrega

9 de junio de 2025

Índice

Índice **2**
Introducción **2**
Marco Teórico **3**
Notación Big O 3
Algoritmos de Búsqueda 4
Algoritmos de Ordenamiento 4
Caso Práctico **5**
Metodología Utilizada **6**
Resultados Obtenidos **7**
Análisis de Resultados 7
Algoritmos de Ordenamiento 7
Algoritmos de Búsqueda 8
Conclusiones **9**
Bibliografía **11**
Anexo **11**

Introducción

El presente informe surge de la decisión conjunta entre los estudiantes Nicolás Arrastía y Mariano Astorga de abordar el estudio de algoritmos de búsqueda y ordenamiento. Esta elección se fundamenta en la relevancia que estos algoritmos poseen dentro del campo de la programación, especialmente en lo que respecta a la eficiencia en la manipulación y recuperación de datos.

Como futuros técnicos universitarios en programación, comprender el funcionamiento interno de estos algoritmos no solo permite optimizar recursos y mejorar el rendimiento de las aplicaciones, sino que también sienta las bases para abordar estructuras más complejas como árboles, grafos o bases de datos. El principal objetivo de este trabajo es analizar los distintos algoritmos de búsqueda y ordenamiento, evaluando su comportamiento, ventajas y limitaciones, a través de una aplicación práctica que demuestre su aplicabilidad en un escenario real o simulado.

Marco Teórico

Notación Big O

La notación Big O describe cómo crece el uso de recursos (tiempo o memoria) de un algoritmo a medida que aumenta el tamaño de la entrada, denotado como “n”. Se enfoca en el caso más costoso (peor caso), aunque también se usa para el caso promedio o el mejor caso. Descarta detalles como constantes o factores menores, porque lo importante es el comportamiento a gran escala.

Permite analizar el tiempo de ejecución de un algoritmo en función del tamaño de la entrada. Esencial para comparar su eficiencia:

- O(1): constante : Tiempo constante, independiente del tamaño. Es lo más eficiente.
- O(log n): logarítmica : Crece lentamente como en la busqueda binaria.
- O(n): lineal: Crece proporcional al tamaño de los datos.
- O(n log n): casi lineal.
- O(n^2): cuadrática: Tipico en algoritmos de fuerza bruta es menos eficiente con grandes volumenes de datos.

La notación Big O ayuda a elegir el algoritmo adecuado según el tamaño de los datos y los requisitos de rendimiento.

(Para el Guion: La notacion big o es una herramienta matematica que nos permite analizar el crecimiento de un algoritmo a medida que crecen los datos a analizar

La notación Big O es esencial para evaluar y comparar la eficiencia de algoritmos. Al analizar algoritmos de búsqueda y ordenamiento en Python, Big O permite identificar cuellos de botella y seleccionar la mejor solución según el contexto. Comprender su aplicación práctica, como en

los ejemplos de búsqueda secuencial y binaria, es clave para optimizar el desarrollo de software.)

Algoritmos de Búsqueda

La búsqueda es una operación fundamental en informática que consiste en localizar un valor específico dentro de una estructura de datos. En Python, se pueden implementar distintos algoritmos de búsqueda, entre los cuales se destacan:

- **Búsqueda lineal (secuencial):** Recorre todos los elementos hasta encontrar el objetivo. Su complejidad es O(n). Es ineficiente para listas grandes
- **Búsqueda binaria:** Solo funciona en listas ordenadas, y reduce el espacio de búsqueda a la mitad en cada iteración. Su complejidad es O(log n).

Aplicaciones para algoritmos de búsqueda:

- Búsqueda de palabras clave
- Búsqueda de un archivo en un sistema de archivos.
- Búsqueda de un registro en una base de datos
- Búsqueda de la ruta más corta en un gráfico
- Búsqueda de soluciones a problemas de optimización

Al comprender los distintos algoritmos y cómo utilizarlos, podemos mejorar el rendimiento y la eficiencia de nuestros programas.

Algoritmos de Ordenamiento

El ordenamiento organiza los datos según algún criterio , lo cual suele ser necesario antes de aplicar una búsqueda binaria o para mejorar la legibilidad y procesamiento posterior. Entre los más conocidos:

- **Bubble Sort:** Intercambia elementos adyacentes si están en el orden incorrecto. $O(n^2)$.
- **Selection Sort:** Busca el menor y lo coloca al inicio repetidamente. $O(n^2)$.

- **Insertion Sort:** Inserta elementos en la posición correcta en una lista ordenada. O(n^2).
- **Merge Sort:** Divide la lista en mitades, las ordena recursivamente y las combina. O(n log n).
- **Quick Sort:** Divide según un pivote y ordena recursivamente. O(n log n) en el mejor caso.

La importancia de los algoritmos de ordenamiento son fundamentales debido a su impacto en la eficiencia, funcionalidad y optimización en sistemas de datos. Es importante en la optimización de procesamiento de datos y tiene impacto en muchos otros campos como la inteligencia artificial

Caso Práctico

Para llevar a cabo el caso práctico, se desarrolló un programa en Python con el objetivo de analizar el comportamiento de distintos algoritmos de ordenamiento y búsqueda. El procedimiento consistió en los siguientes pasos:

1. **Generación de datos**: Se creó una lista aleatoria compuesta por *n* números enteros, siendo *n = 20* en este caso. Además, se generó un número entero aleatorio adicional, utilizado como valor de búsqueda en los algoritmos correspondientes.
2. **Ejecución de algoritmos**:
    - Se aplicaron los algoritmos de ordenamiento **Bubble Sort**, **Insertion Sort**, **Selection Sort**, **Merge Sort** y **Quick Sort** para ver si coincidía la teoría con la ejecución del código.
    - Posteriormente, se ejecutaron los algoritmos de búsqueda **Lineal** y **Binaria**, utilizando la lista previamente ordenada para el segundo caso.
3. **Medición de resultados**:

    Durante la ejecución de cada algoritmo, se registraron dos métricas principales: el **tiempo de ejecución** y la **cantidad de comparaciones realizadas** hasta completar el proceso tanto para los algoritmos de ordenamiento como para los de búsqueda.

4. **Visualización**:

   Finalmente, se utilizaron bibliotecas gráficas de Python para generar gráficos comparativos que muestran el desempeño de cada algoritmo en términos de tiempo y comparaciones.

Metodología Utilizada

El proceso de realización del trabajo se llevó a cabo mediante una serie de fases cuidadosamente planificadas, descritas a continuación:

1. **Búsqueda y recopilación de información teórica**: Se realizó una exhaustiva revisión de fuentes confiables, incluyendo libros, artículos científicos y documentación técnica, con el objetivo de construir una base sólida de conocimiento sobre los temas y algoritmos relevantes para el proyecto.
2. **Desarrollo e implementación de algoritmos en Python**: Se procedió a la codificación de los algoritmos estudiados, utilizando el lenguaje de programación Python debido a su versatilidad y amplia disponibilidad de bibliotecas. Esta etapa incluyó la creación de scripts optimizados y modulares para garantizar un código eficiente y mantenible.
3. **Ejecución de pruebas con diversos conjuntos de datos**: Se llevaron a cabo experimentos utilizando diferentes conjuntos de datos, seleccionados cuidadosamente para abarcar una amplia gama de escenarios y condiciones. Esta fase permitió evaluar el comportamiento y la robustez de los algoritmos implementados.
4. **Análisis y documentación de resultados**: Los resultados obtenidos de las pruebas fueron registrados meticulosamente, incluyendo métricas de desempeño y observaciones cualitativas. Además, se validó la funcionalidad de los algoritmos comparando los resultados con los esperados, ajustando parámetros cuando fue necesario.
5. **Redacción del informe final y preparación de anexos**: Se elaboró un informe detallado que sintetiza el proceso, los hallazgos y las conclusiones del proyecto. Asimismo, se incluyeron anexos con información complementaria, como gráficos, tablas de datos y fragmentos de código relevantes, para respaldar los resultados presentados.

6. **Creación del video explicativo:** Se produjo un video explicativo con el objetivo de presentar de forma clara y visual los aspectos más relevantes del trabajo. En él se describen los algoritmos implementados, se muestran ejemplos de ejecución y se resumen las conclusiones obtenidas, facilitando así la comprensión del proyecto.

Resultados Obtenidos

Análisis de Resultados

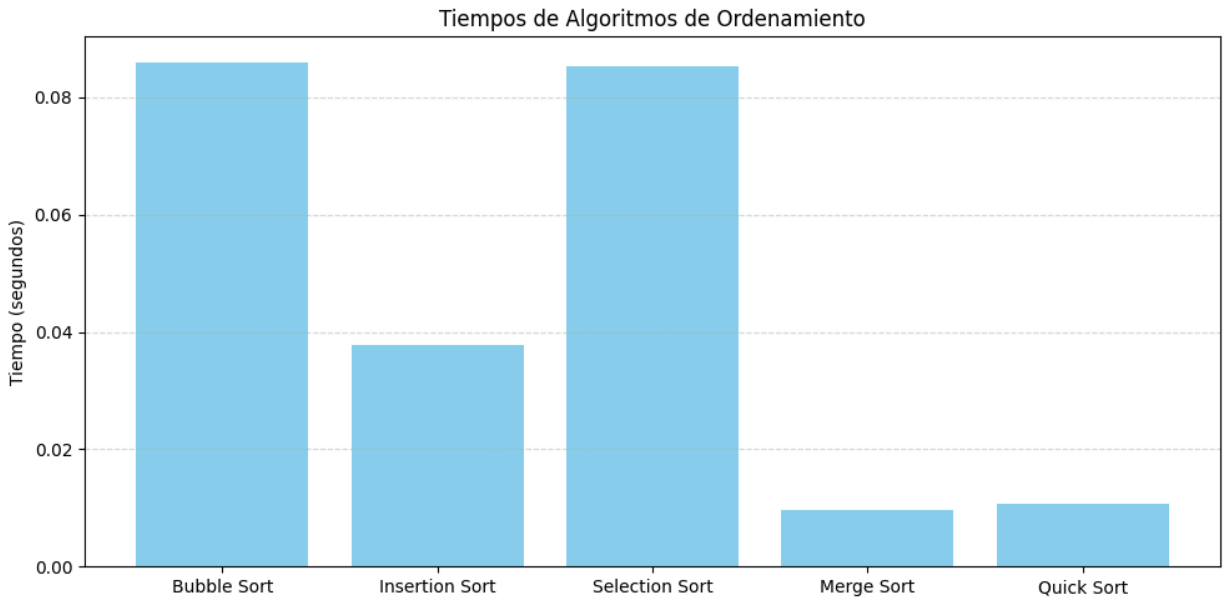
Tras la ejecución y evaluación de distintos algoritmos de ordenamiento y búsqueda, se obtuvieron resultados significativos en cuanto al tiempo de ejecución y la cantidad de comparaciones realizadas por cada uno. Estos datos permiten comprender mejor la eficiencia relativa de cada método en distintos contextos.

Algoritmos de Ordenamiento

Se probaron cinco algoritmos de ordenamiento multiples veces para corroborar la consistencia de los resultados. Los algoritmos fueron los siguientes: Bubble Sort, Insertion Sort, Selection Sort y Merge Sort. A continuación se detallan los resultados obtenidos:

- **Bubble Sort**: Este algoritmo demostró ser el menos eficiente en cuanto a tiempo de ejecución y número de comparaciones, especialmente con listas de gran tamaño. Al ser un algoritmo de complejidad cuadrática ($O(n^2)$), su rendimiento decrece rápidamente con el aumento de datos.
- **Insertion Sort**: Tuvo un rendimiento levemente superior al de Bubble Sort en listas parcialmente ordenadas, aunque también muestra un comportamiento cuadrático en el peor de los casos.
- **Selection Sort**: Si bien realiza un número fijo de comparaciones sin importar el estado inicial de la lista, su tiempo de ejecución no mejora frente a Insertion Sort. Este comportamiento también responde a su complejidad $O(n^2)$.

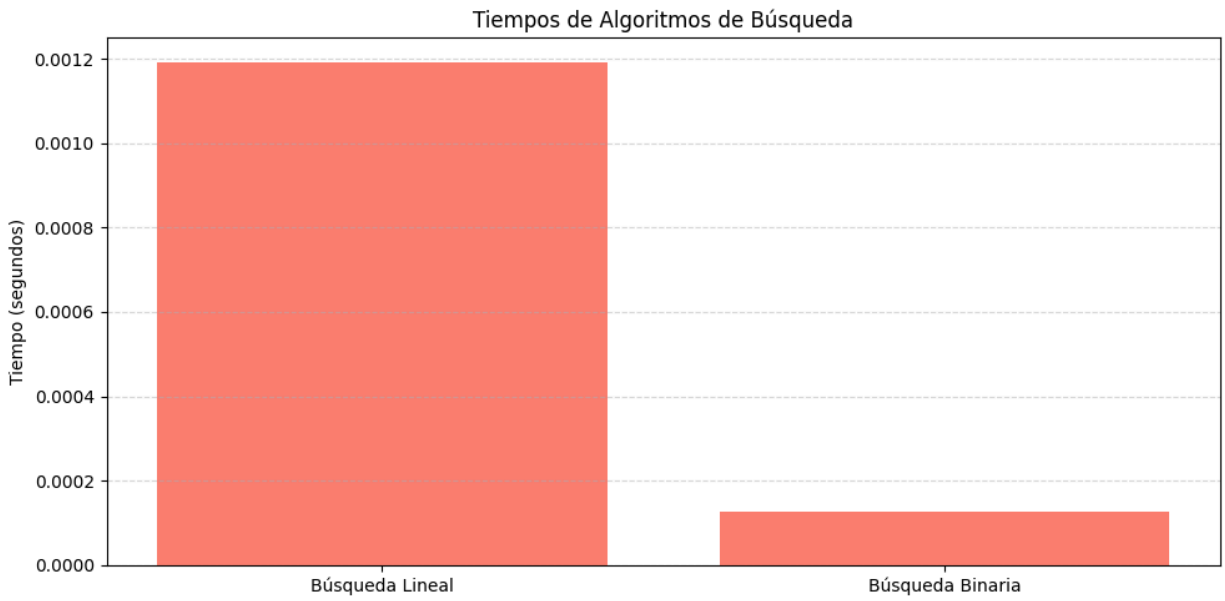
- **Merge Sort**: Fue el algoritmo más eficiente entre los probados, tanto en tiempo como en número de comparaciones. Su enfoque divide y conquista, junto a su complejidad O(n log n), lo convierte en una mejor opción para listas de mayor tamaño.



Algoritmos de Búsqueda

Se evaluaron dos métodos: Búsqueda Lineal y Búsqueda Binaria.

- **Búsqueda Lineal**: Mostró una eficiencia aceptable en listas pequeñas, pero su rendimiento disminuyó notablemente a medida que se incrementaba el tamaño de la estructura. Esto se debe a que recorre elemento por elemento hasta encontrar el objetivo o finalizar la lista.
- **Búsqueda Binaria**: Solo aplicable en listas ordenadas, la búsqueda binaria destacó por su baja cantidad de comparaciones y tiempos mínimos. Su complejidad O(log n) la posiciona como la opción más eficiente cuando la condición del orden se cumple.



Los resultados obtenidos confirman las predicciones teóricas respecto al rendimiento de cada algoritmo. Mientras los métodos como Bubble, Insertion y Selection Sort presentan limitaciones con grandes volúmenes de datos, Merge Sort y la Búsqueda Binaria se consolidan como soluciones más escalables y eficientes. Las imágenes que acompañan el análisis muestran en detalle los tiempos de ejecución por cada algoritmo.

Conclusiones

Tras el desarrollo del proyecto enfocado en los algoritmos de búsqueda y ordenamiento implementados en Python, se pueden extraer las siguientes conclusiones:

1. **Eficiencia y adaptabilidad de los algoritmos**: Los algoritmos de búsqueda (como la búsqueda lineal y binaria) y de ordenamiento (como burbuja, selección, inserción, merge sort y quicksort) presentan diferentes niveles de eficiencia dependiendo del contexto. Por ejemplo, la búsqueda binaria es significativamente más rápida que la lineal, pero requiere que los datos estén ordenados previamente. De manera similar, algoritmos como merge sort y quicksort superan en rendimiento a los métodos más simples como el ordenamiento por burbuja en conjuntos de datos grandes, aunque estos últimos pueden ser útiles para listas pequeñas debido a su simplicidad.

2. **Ventajas de Python en la implementación**: Python, gracias a su sintaxis clara y su extensa colección de bibliotecas, facilita la implementación y prueba de estos algoritmos. La flexibilidad del lenguaje permitió desarrollar código modular y reutilizable, lo que agilizó las fases de codificación y depuración. Además, bibliotecas como time y random fueron clave para medir el rendimiento y generar conjuntos de datos variados.
3. **Importancia de la selección del algoritmo**: Las pruebas con diferentes conjuntos de datos demostraron que la elección del algoritmo depende del tamaño del conjunto de datos, su estructura y los requisitos de la aplicación. Por ejemplo, quicksort mostró un rendimiento promedio excelente, pero en casos patológicos (listas casi ordenadas o inversas) su eficiencia puede degradarse, haciendo preferible merge sort en ciertos escenarios.
4. **Validación práctica de la teoría**: La implementación y evaluación práctica de los algoritmos corroboraron los conceptos teóricos estudiados. Los resultados obtenidos en las pruebas, como los tiempos de ejecución y la precisión de las búsquedas, reflejaron las complejidades temporales esperadas ($O(n)$, $O(\log n)$, $O(n^2)$, etc.), lo que refuerza la importancia de comprender las bases teóricas antes de aplicarlas.
5. **Relevancia en aplicaciones reales**: Los algoritmos de búsqueda y ordenamiento son fundamentales en numerosas aplicaciones del mundo real, como bases de datos, motores de búsqueda y sistemas de recomendación. Este proyecto destacó cómo su correcta implementación en Python puede optimizar procesos computacionales, mejorando la eficiencia y escalabilidad de las soluciones.
6. **Oportunidades de mejora**: Aunque los algoritmos implementados cumplieron con los objetivos, se identificaron áreas de mejora, como la optimización de memoria en algoritmos recursivos (por ejemplo, quicksort) o la incorporación de técnicas avanzadas, como algoritmos híbridos o paralelos, para manejar conjuntos de datos masivos.

Bibliografía

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

Insertion Sort y Bubble Sort

https://www.youtube.com/watch?v=TZRWRjq2CAg&ab_channel=udiprod

Quick Sort

https://www.youtube.com/watch?v=Vtckgz38QHs&t=3s&ab_channel=BroCode

Merge Sort

https://www.youtube.com/watch?v=3j0SWDX4AtU&ab_channel=BroCode

Anexo

- Repositorio en GitHub: https://github.com/NicolasArrastia/UTN-Prog1-Integrador
- Video explicativo: https://youtu.be/YrDKrxED9jg
- Diapositivas link: https://docs.google.com/presentation/d/1twqEFlCFETRsPkHLpb8wePxwfmU3T2kDwWjIIIOvm00/edit?usp=drive_link