Trabajo Práctico Integrador

Tema: Algoritmos de Búsqueda y Ordenamiento

“Implementación y análisis de algoritmos de ordenamiento y búsqueda en Python”

Alumnos

Santiago Octavio Varela (santiagov.linked@gmail.com) ,

Ximena Maribel Sosa (ximenasosa44@gmail.com)

(Ambos Comisión 22)

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación I

Docente Titular

Nicolas Quirós Ricci

Docente Tutor

Matias Santiago Torres

Fecha de entrega

9 de Junio de 2025

Tabla de contenido

1. Introducción

El presente trabajo se centra en el análisis y la implementación de algoritmos de ordenamiento y búsqueda, dos herramientas fundamentales en el desarrollo de software y en la optimización de procesos computacionales. La elección de este tema responde a la necesidad de comprender en profundidad cómo la organización previa de los datos puede influir significativamente en la eficiencia de los algoritmos de búsqueda, aspecto que es de importancia en múltiples aplicaciones prácticas y sistemas informáticos.

Desde la perspectiva del área de estudio, el tema resulta de gran relevancia ya que integra conceptos teóricos con aplicaciones prácticas en programación. La utilización de algoritmos como el Bubble Sort para ordenar los datos, mientras que la búsqueda binaria y lineal para localizarlos ofrece un claro ejemplo de cómo la eficiencia algorítmica depende de la correcta preparación y estructuración de la información. Este enfoque permite a los técnicos en programación consolidar conocimientos sobre la complejidad computacional y el manejo de estructuras de datos, competencias esenciales en su formación profesional.

El objetivo principal del trabajo es desarrollar un programa en Python que, mediante la combinación de técnicas de ordenamiento y búsqueda, evidencie la mejora en el rendimiento y la organización de la información. Además, se busca que el proyecto fomente la capacidad de análisis crítico en la selección y comparación de diferentes algoritmos, incentivando la exploración de estrategias adicionales como la medición de tiempos de ejecución, la visualización paso a paso del proceso y la validación de entradas. Estos elementos no solo contribuirán a la optimización de tareas computacionales, sino que también brindarán una base sólida para el desarrollo de soluciones más complejas en futuras aplicaciones que los integrantes puedan realizar.

En síntesis, el trabajo se propone alcanzar la integración de conceptos fundamentales de la programación con prácticas de experimentación y análisis comparativo, herramientas esenciales para la formación de técnicos capaces de enfrentar desafíos en el ámbito del desarrollo de software y la optimización de procesos.

2. Marco Teórico

Los algoritmos de búsqueda y ordenamiento son fundamentales en el campo de la programación, ya que permiten localizar elementos específicos dentro de un conjunto (búsqueda) y organizar los datos según un criterio definido (ordenamiento).

Algoritmos de Ordenamiento

Empezando con el ordenamiento, el objetivo del mismo es reorganizar un conjunto de elementos bajo un criterio específico, como orden numérico ascendente o descendente, o alfabético:

- **Bubble Sort** (ordenamiento burbuja): compara elementos adyacentes e intercambia su posición si están desordenados. Su complejidad es $O(n^2)$, por lo que resulta ineficiente en listas grandes, aunque útil con fines educativos (W3Schools, s.f.), una de las razones por las cuáles se eligió este tipo de ordenamiento para nuestro caso.
- **Insertion Sort:** inserta cada nuevo elemento en su lugar correcto dentro de la porción ya ordenada de la lista. Es intuitivo y adecuado para listas pequeñas o parcialmente ordenadas, también con una complejidad $O(n^2)$ en el peor caso. (Python Software Foundation, 2024)
- **Selection Sort:** busca el menor valor y lo coloca en la primera posición, repitiendo el proceso con los elementos restantes. Aunque conceptualmente simple, también presenta $O(n^2)$ de complejidad, siendo más lento que los anteriores. (GeeksforGeeks, s.f.)
- **Quicksort:** implementa la estrategia de “divide y vencerás”, eligiendo un pivote y dividiendo la lista en dos sublistas menores y mayores. Es uno de los algoritmos más eficientes, con una complejidad promedio de $O(n \log n)$, aunque en el peor caso puede alcanzar $O(n^2)$ si el pivote es mal seleccionado. (Cormen et al., 2009)

Los algoritmos elegidos dentro de las listas previamente mencionadas, que fueron solo Bubble Sort para ordenamiento mientras que para búsqueda son búsqueda binaria (solo si la lista se ordena previamente) y búsqueda lineal, fueron implementados y probados en Python 3. Se utilizaron listas como estructura principal, junto con los módulos random (para generación de listas aleatorias) y time (para medir el tiempo de ejecución). La documentación

oficial de Python detalla el funcionamiento de estas herramientas. (Python Software Foundation, 2024)

Comprender estos algoritmos es fundamental para elegir el método adecuado según el contexto, evaluando tamaño de los datos, orden previo, y la eficiencia deseada.

Cuadro comparativo entre los diferentes tipos de algoritmos de ordenamiento[1]

		Burbuja	Burbuja Bidireccional	QuickSort
Funcionamiento		Revisa cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado.	Mejora del método burbuja Ordena al mismo tiempo por ambas puntas del vector o arreglo.	Algoritmo basado en divide y vencerás, que permite ordenar una X cantidad de elementos en un tiempo proporcional
Orden de complejidad	Peor Caso	$O(n^2)$	$O(n^2)$	$O(n^2)$
	Caso Promedio	$O(n^2)$	$O(n^2)$	O(n log n)
	Mejor Caso	O(n)	O(n)	O(n log n)
Complejidad	Tiempo	0,0040s	0,0030s	0,0010s
	Espacio	307 bytes	1.135 bytes	971 bytes
Ventaja		Sencillo, eficaz.	Más rápido que su antecesor burbuja, ya que trabaja desde ambas puntas.	Muy rápido, no necesita memora extra
Desventaja		Consume muchos recursos.	Consume muchos recursos (menos que burbuja)	Implementación complicada.

[1] Fuente: https://es.slideshare.net/slideshow/cuadro-comparativo-algoritmos-de-ordenamiento-35503780/35503780#1

		HeapSort	ShellSort	Inserción
Funcionamiento		Consiste en almacenar todos los elementos en un solo nodo o punta, para luego extraer esa punta hasta ordenar todos los datos. Usa un árbol binario para poder dar forma al proceso a la hora de ordenar.	Mejora el ordenamiento por inserción. Compara elementos separados por un espacio aun mayor y los ordena.	Consiste en ir comparando los elementos y ordenando a medida que se avanza por el arreglo.
Orden de complejidad	Peor Caso	O(n log n)	$O(n^2)$	$O(n^2)$
	Caso Promedio	O(n log n)	-	$O(n^2)$
	Mejor Caso	O(n log n)	O(n log n)	O(n)
Complejidad	Tiempo	0,0010s	0,0010s	0,0040s
	Espacio	1.225bytes	722bytes	1.150bytes
Ventaja		Desempeño tan bueno como QuickSort.	Trabaja bien con arreglos pequeños, no necesita memoria extra.	Fácil de implementar, requiere poca memoria
Desventaja		Complejo de programar.	No funciona con arreglos mayores a 5000 elementos.	Lento, muchas comparaciones

La notación O(n) como parte del diseño de este trabajo

Al analizar algoritmos, una de las principales preocupaciones es cuánto tiempo pueden tardar en ejecutarse ante diferentes tamaños de entrada. Como hemos visto en los textos de cátedra (Tecnicatura Universitaria en Programación, 2025) se suele considerar el peor caso posible para hacer esta medición, es decir, el escenario donde el algoritmo requiere la mayor cantidad de pasos para completarse, siendo O(n) el peor caso. Este enfoque resulta útil porque garantiza que más allá de los datos que estemos manejando, el rendimiento del programa no superará cierto límite. De esta manera, se pueden establecer comparaciones y tomar decisiones fundamentadas sobre qué algoritmo utilizar según la situación.

La notación O(n), conocida como notación “Big O”, es una forma estandarizada de describir la eficiencia de un algoritmo en función del tamaño de la entrada. Esta notación permite estimar cuánto se incrementan los recursos necesarios principalmente tiempo de ejecución o uso de memoria a medida que crece la cantidad de datos, sin depender de factores externos (lenguaje, hardware utilizado, etc.)

En este trabajo, la notación “Big O” fue una herramienta clave para justificar decisiones de diseño. Al implementar tanto búsqueda lineal como búsqueda binaria, fue tenido en cuenta

que la primera crece linealmente con el tamaño de la lista (O(n)), mientras que la segunda lo hace logarítmicamente (O(log n)). Esta diferencia fue fundamental para definir cuándo y cómo habilitar cada una, ya que la búsqueda binaria solo se ofrece si la lista está ordenada y su eficiencia únicamente puede aprovecharse en ese contexto.

Lo mismo ocurre con el algoritmo de ordenamiento elegido, Bubble Sort, cuya complejidad es $O(n^2)$. Aunque ya hemos visto que no es el más eficiente para grandes volúmenes de datos, se optó por él por su simplicidad e importancia didáctica. Aun así, el programa permite medir el tiempo de ejecución real de cada algoritmo, facilitando una comparación entre lo que predice la notación O(n) y lo que ocurre en la práctica. En consecuencia, la notación “Big O” no solo estructura la teoría detrás de los algoritmos utilizados, sino que también guió decisiones concretas en el diseño y la interacción del programa.

Algoritmos de búsqueda

Ahora nos centraremos sintéticamente en los algoritmos de búsqueda utilizados para este trabajo. Según Cormen et al. (2009), un algoritmo de búsqueda es aquel que permite localizar un elemento objetivo dentro de una estructura de datos, devolviendo su posición si se encuentra. Existen diversos tipos de algoritmos de búsqueda, siendo las más comunes la búsqueda lineal y la búsqueda binaria, aunque no siempre los más aptos para todos los casos.

Búsqueda lineal

Es un método sencillo que recorre cada elemento de una lista hasta hallar el valor deseado. Tiene una complejidad temporal de O(n), lo que la hace adecuada para listas pequeñas o no ordenadas. En el peor de los casos, el elemento buscado está al final de la lista o no se encuentra (GeeksforGeeks, s.f.).

En el programa, este tipo de búsqueda está disponible tanto cuando la lista ha sido ordenada previamente como cuando no. Esta decisión se tomó para permitir una comparación directa entre ambos métodos y para garantizar un marco de condiciones controladas al momento de evaluar el rendimiento.

Búsqueda binaria

Es mucho más eficiente (O(log n)), pero requiere que la lista esté previamente ordenada. Su funcionamiento se basa en comparar el valor central de la lista con el elemento buscado, descartando sistemáticamente la mitad que no contiene el objetivo (Cormen et al., 2009).

En el programa, solo se habilita esta opción si el usuario decide ordenar la lista. Esta restricción permite garantizar su correcto funcionamiento y al mismo tiempo evidencia, en la práctica, cómo las condiciones estructurales de los datos influyen directamente en qué algoritmos pueden utilizarse.

Cabe señalar que la búsqueda binaria admite múltiples implementaciones, siendo las más comunes la versión iterativa y la recursiva (DataCamp, s.f.). En este trabajo se optó por la forma iterativa por su mayor eficiencia en el uso de memoria, estabilidad frente a listas extensas y mayor facilidad de depuración paso a paso. En la otra forma de aplicación, la versión recursiva ofrece la resolución del problema en subespacios cada vez más reducidos. La comparación entre ambas formas puede considerarse una extensión futura del presente trabajo.

3. Caso Práctico

Descripción del problema a resolver

En el presente trabajo práctico se plantea el desarrollo de un programa en Python que permita ordenar e identificar elementos dentro de una lista de datos numéricos. La finalidad del proyecto es comprender el funcionamiento de algoritmos clásicos de ordenamiento y búsqueda, así como analizar sus características y desempeño en situaciones controladas.

El programa implementará un flujo que combine el algoritmo de ordenamiento Bubble Sort con los algoritmos de búsqueda binario y lineal, permitiendo al usuario observar la importancia de organizar los datos antes de ejecutar búsquedas eficientes, además de una comparativa entre los dos tipos de búsqueda. A partir de este enfoque, se busca reforzar dos conceptos clave en programación: la preparación de estructuras de datos y la selección de estrategias algorítmicas adecuadas.

El problema central consiste en permitir al usuario ingresar una cantidad (solamente se le pregunta por la cantidad) de números de forma aleatoria y en formato lista. Luego realizar dos operaciones principales:

- Ordenamiento de la lista o sin ordenar: El programa le solicitará al usuario si quiere ordenar la lista utilizando el algoritmo de Bubble Sort, el cual, aunque no es el más eficiente para grandes volúmenes, es pedagógicamente útil por su simplicidad y su fácil visualización paso a paso. En caso de que no quiera ordenarla, se le dará la opción de usar solo la búsqueda lineal o no realizar búsquedas.
- Búsqueda de un elemento dentro de la lista ordenada: Se utilizará la búsqueda binaria, una técnica eficiente para listas previamente ordenadas como condición excluyente y con complejidad logarítmica, o bien la búsqueda lineal previamente mencionada, dándole a elegir al usuario entre ambas en caso de listas ordenadas.

Entonces, no se podrá elegir entre otras opciones de ordenamiento por fuera de Bubble Sort, pero sí se podrá elegir entre opciones de búsqueda binaria y búsqueda lineal en el programa previsto. Esta es una decisión estrictamente pedagógica, que se ha decidido en función de mantener un entorno que los estudiantes pudiesen controlar en relación a los recursos, el

tiempo previsto y los objetivos planteados para este trabajo desde las consignas propuestas por la cátedra.

Con esta solución, es posible observar uno de los factores más relevantes que ha solicitado dentro de la consigna general, que es la de comparar resultados en función de la selección de al menos dos de los algoritmos de búsqueda presentes en el material teórico. A su vez, si bien solo se usa Bubble Sort en el aspecto del ordenamiento, el hecho de ordenar la lista o no ordenarla también brinda una comparativa acerca de la importancia del ordenamiento, más allá de la eficiencia relativa que pueda tener o no Bubble Sort respecto a otras formas de ordenamiento.

Además, se incorporarán las siguientes extensiones para enriquecer la experiencia del usuario y profundizar en el análisis computacional:

- Medición de tiempo de ejecución: se mostrará el tiempo que tarda cada algoritmo en ejecutarse para evidenciar su eficiencia relativa.
- Visualización paso a paso del ordenamiento: se presentará cómo se transforma la lista en cada iteración del algoritmo de ordenamiento.
- Validación de entradas: se incluirán controles para garantizar que los datos ingresados sean apropiados para los algoritmos a utilizar (por ejemplo, no permitir búsqueda binaria en una lista sin ordenar).

A través de esta implementación, el trabajo apunta a ilustrar de forma práctica la relación entre la eficiencia algorítmica y la estructura de los datos, reforzando conceptos fundamentales para la resolución de problemas computacionales.

Validación del funcionamiento

Para validar el correcto funcionamiento del programa, se realizó una ejecución completa del mismo, verificando cada etapa del proceso: generación de números aleatorios en una lista, ordenamiento mediante Bubble Sort o sin ordenar y búsqueda binaria o búsqueda lineal de un elemento, dependiendo de qué opción se elija. Cada una estas etapas pueden verse a lo largo del video explicativo y la breve demostración que se encuentra en el repositorio del proyecto[2].

[2]URL del repositorio: https://github.com/santiagovOK/programacion-uno_integrador_VarelaSosa

El primer paso consiste en solicitar al usuario la cantidad de números aleatorios a generar. Esta entrada se valida para asegurar que sea un valor entero positivo. Luego, se crea una lista de enteros aleatorios sin repetición, cuya longitud coincide con el valor ingresado. Esta lista es mostrada al usuario para su revisión.

A continuación, el programa ofrece la posibilidad de ordenar dicha lista mediante el algoritmo Bubble Sort o no ordenar la lista. Si el usuario acepta ordenar con Bubble Sort, se registra el tiempo que toma el proceso y, opcionalmente, se pueden visualizar los pasos intermedios de cada iteración. El resultado final incluye la lista ordenada y el tiempo de ejecución en segundos. Por otra parte, la opción de no ordenar la lista lleva directamente al usuario a la pregunta sobre si quiere buscar un elemento en la lista.

Una vez pasada la etapa de ordenamiento, se habilita la opción de buscar un número. El usuario puede ingresar un valor manualmente o dejar que el programa seleccione uno aleatoriamente desde la lista, ya sea si ordenó o no previamente la lista. En el caso de que esté ordenada, el usuario puede elegir entre búsqueda lineal o búsqueda binaria. Esto habilita a la comparación principal del programa, siendo que el punto de partida es el mismo para ambos tipos de búsqueda: la lista está ordenada. Se mide también el tiempo que toma esta operación, y se indica si el número fue encontrado o no, junto con su posición para ambos tipos de búsqueda. En cambio, si la decisión fue no ordenar la lista previamente y se ejecuta la búsqueda, esta podrá ser búsqueda lineal dentro del programa como única opción.

Por último, el programa muestra un resumen con los tiempos de ordenamiento y búsqueda. La ejecución concluye con un mensaje de finalización, indicando que el programa se desarrolló correctamente sin errores visibles, y que cada módulo cumple con su función esperada dentro del flujo general. Podrán ver el código del programa completo en el anexo de este informe y con más detalle en el repositorio ya mencionado.

4. Metodología Utilizada

Investigación previa

Antes de comenzar con el desarrollo del trabajo, fue realizada una revisión teórica sobre algoritmos de búsqueda y ordenamiento. Para ello consultamos en fuentes académicas como Introduction to Algorithms de Cormen et al. (2009), la documentación oficial de Python (Python Software Foundation, 2024), y sitios educativos como GeeksforGeeks, W3Schools y Real Python. También analizamos las explicaciones brindadas por el docente en los videos de clase, lo cual permitió comprender las diferencias entre cada algoritmo, su complejidad temporal y los casos ideales para su uso.

Además, se investigó sobre algunas funciones y métodos de Python que, si bien quizás fueron mencionados a lo largo de las unidades de la materia, aún no habíamos explorado en profundidad. En general, todos ellos se encuentran fácilmente usando el buscador de la documentación oficial de Python citada en el párrafo anterior.

Uno de ellos es strip(), que elimina espacios en blanco al principio y al final de una cadena, permitiendo procesar entradas del usuario de forma más precisa. En combinación con isdigit(), que verifica si una cadena representa un número entero, se logra una validación robusta de datos ingresados. Por otra parte, la función random.sample() fue utilizada para generar listas de números sin repeticiones y permitió ejecutar casos de prueba de manera controlada. Además, se empleó time.time() para capturar el tiempo de ejecución de los algoritmos, algo nuevo para nosotros y útil para evidenciar diferencias de eficiencia. Finalmente, fue utilizado formateo de cadenas como :.6f para mostrar los tiempos con precisión, reforzando la presentación clara de resultados. Estas incorporaciones ampliaron el repertorio de herramientas de los estudiantes para el uso de Python.

Etapas de diseño y prueba del código

El desarrollo del programa fue realizado en función de dos versiones. Una versión inicial estructurada secuencialmente y una segunda versión con un diseño modular más claro, escalable y al cuál se le puede hacer un mejor seguimiento. Esta segunda versión es la que dió comienzo al repositorio.

En la primera versión, el código fue realizado como un bloque único, teniendo en cuenta los dos ejes principales del problema planteado: una función de ordenamiento con Bubble Sort, y otra de búsqueda con búsqueda binaria. Ambas operaciones principales, además de la entrada de datos solicitada al usuario, se encontraban integradas en una secuencia lineal.

Esta versión era operativa pero presentaba dificultades en términos de mantenimiento y reutilización. Además, la repetición de código y la falta de separación lógica entre las distintas tareas pueden dificultar las pruebas y el análisis de cada componente de forma individual, lo que también podía complicar la descripción y explicación del trabajo de parte de los integrantes en el video de presentación a realizar.

A partir de una revisión de esta estructura inicial, se optó por una refactorización del programa con énfasis en la modularidad, como se viene viendo en parte durante las clases de la asignatura Programación I. Las funcionalidades principales se dividieron en funciones específicas, buscando ejecutarlas dentro de main() como programa principal. Esta nueva organización permitió no solo mejorar la legibilidad del código, sino también facilitar su prueba y validación y simplificación por separado.También se incorporaron mecanismos de validación de entrada y medición de tiempos de ejecución para ambos algoritmos, lo cual permitió comparar su eficiencia y verificar su correcto funcionamiento.

Luego de la modularización y pensando en la presentación de este trabajo, se decidió simplificar aún más el código en general retomando función por función, revisando la claridad de las estructuras creadas, la denominación de las variables y la complejidad en general. Tras una primera revisión, fue simplificada la interacción inicial con el usuario, eliminando la opción de ingresar una lista personalizada y optando por una lista aleatoria de números enteros. Esto se debe a que la lista personalizada no aportaba valor significativo al problema central que busca tratar este trabajo y terminaba complejizando innecesariamente los primeros pasos de ejecución del programa.

En conclusión, la evolución del programa demuestra la importancia del diseño iterativo y de la modularidad en el desarrollo de software, especialmente en contextos educativos donde la comprensión del flujo lógico y la claridad del código son fundamentales.

Herramientas y recursos utilizados

Para la elaboración de este trabajo práctico se utilizaron diversas herramientas que facilitaron tanto la organización del grupo como la implementación del programa en Python. La redacción colaborativa del informe se llevó a cabo mediante **Google Docs**, permitiendo a los integrantes del equipo trabajar simultáneamente y mantener un registro continuo de los avances y ajustes realizados.

En lo que respecta al entorno de desarrollo, se utilizó **Visual Studio Code**, un editor de código ampliamente difundido por su versatilidad y disponibilidad de extensiones. Esta herramienta fue empleada para escribir el código fuente, realizar pruebas y organizar el proyecto de forma modular. También se aprovechó su sistema de terminal integrada para ejecutar directamente el programa y verificar su correcto funcionamiento.

El control de versiones se realizó a través de **Git**, junto con la plataforma **GitHub** para alojar el repositorio del proyecto. Esto permitió mantener un seguimiento ordenado de las modificaciones al código, así como compartir y sincronizar los avances del equipo de manera eficiente.

Reparto de tareas y trabajo colaborativo

El desarrollo del trabajo práctico se realizó de manera colaborativa, con una distribución clara de responsabilidades entre ambos integrantes del equipo, permitiendo un avance ordenado y complementario en todas las etapas del proyecto.

Santiago se encargó de la búsqueda de fuentes adicionales más allá del material provisto por la cátedra, lo cual permitió enriquecer el enfoque teórico y técnico del trabajo. Asumió la redacción inicial de la introducción, la descripción del problema y el detalle de las herramientas utilizadas, así como también una revisión general de estilo, ortografía y coherencia textual. En términos técnicos, definió la estructura inicial del repositorio priorizando la modularización del programa, y fue responsable de ampliar y refinar el código: simplificó funciones, renombró variables para mejorar su claridad y reorganizó la lógica del programa. También desarrolló el guión individual para la presentación, editó el video y lo publicó. Finalmente, redactó la primera parte de la sección de validación del código y participó activamente en la elaboración de las conclusiones.

Ximena, por su parte, se encargó del borrador inicial del código en forma secuencial, sobre el cual luego se aplicaron mejoras. Profundizó las referencias bibliográficas utilizadas y desarrolló el marco teórico del informe. Elaboró el guión general del video de presentación, sobre el que cada integrante construyó luego su parte. Además, tuvo a cargo la redacción final de la sección de validación del código y la subsección de validación del funcionamiento, asegurando que los resultados puedan observarse claramente. También revisó y organizó el contenido del archivo README.md del repositorio, asegurando que reflejara el funcionamiento del programa, la documentación del proyecto y los enlaces correspondientes. Finalmente, redactó la sección de Resultados Obtenidos.

Esta división de tareas permitió un desarrollo fluido y un producto final integrado, con participación activa y equitativa de ambos miembros del equipo.

5. Resultados Obtenidos

La implementación final del programa permitió experimentar con algoritmos de ordenamiento y búsqueda, observando su comportamiento mediante pruebas controladas. Se verificó que el algoritmo de Bubble Sort ordena correctamente listas de enteros y que la opción de mostrar los pasos intermedios facilita la comprensión de su funcionamiento en cada iteración.

El programa genera listas de forma aleatoria y, hasta aquí, se ha probado con tamaños que variaron entre 5 y 50 elementos. Si bien no le hemos otorgado límites a la cantidad de elementos que se pueden ingresar, el ordenamiento hasta esas cantidades fue exitoso en todos los casos. El tiempo de ejecución, medido con time(), mostró que para listas pequeñas los tiempos son mínimos (milisegundos), mientras que para listas más grandes se incrementan un poco más.

En cuanto a la búsqueda binaria, se confirmó su correcto funcionamiento siempre y cuando las listas estén previamente ordenadas. El programa devuelve la posición del elemento si este se encuentra dentro de la lista y utiliza el valor -1 como indicador de ausencia. Cuando este valor es interpretado por el programa, imprime el mensaje: “Elemento no encontrado”, lo cual proporciona una respuesta clara e intuitiva al usuario.

También se evaluó la búsqueda lineal, comprobando que funciona correctamente tanto en listas ordenadas como desordenadas. Esta búsqueda recorre los elementos uno por uno, por lo que fue útil para comparar su rendimiento frente a la búsqueda binaria. Los resultados mostraron que, aunque es más simple, la búsqueda lineal es menos eficiente en listas grandes, pero sigue siendo una opción válida cuando los datos no están ordenados o la cantidad de datos no es tan alta.

Entre las dificultades enfrentadas es posible incluir que, si bien entre los temas posibles a abordar quizás búsqueda y ordenamiento podría considerarse como uno de los más accesibles, la necesidad de entregar este trabajo a término en relación al poco tiempo disponible podría considerarse como un desafío significativo a la hora de investigar un tema totalmente nuevo para los integrantes de este grupo. Tampoco no fue sencilla, aunque era necesaria, la modularización del código para mejorar su claridad, seguimiento respecto a lo conceptual y la accesibilidad para su presentación de parte nuestra. Es por eso que se optó

por la decisión de acotar el script en Python a un solo tipo de ordenamiento (Bubble Sort) y a dos clases de búsqueda (binaria y lineal) dentro de las existentes.

En conjunto, los resultados obtenidos demuestran la efectividad del programa como herramienta didáctica para explorar conceptos fundamentales de programación y algoritmos.

6. Conclusiones

Es posible concluir que el desarrollo de este trabajo permitió al grupo afianzar conceptos fundamentales de programación, como el diseño modular, el uso de funciones y la implementación de algoritmos considerados "típicos" como Bubble Sort, búsqueda binaria y lineal, aunque no necesariamente los más utilizados en general. Se reforzó cómo estructurar un programa que interactúe con el usuario dentro de la terminal, validando entradas y con condiciones de salida precisas, o bien mostrando resultados de manera clara.

Otro aprendizaje clave fue la medición de tiempos de ejecución y la impresión de iteraciones en la parte de ordenamiento, ya que permite observar cómo la eficiencia de los algoritmos se ve reflejada en la práctica, especialmente al comparar el caso Bubble Sort ($O(n^2)$) con los dos tipos de búsqueda planteados. Además, este tipo de prácticas son importantes para el manejo de errores, una herramienta clave en la depuración del código.

Ya mencionamos que durante el proceso enfrentamos algunos desafíos, como la necesidad de un aprendizaje acelerado del tema frente a la fecha límite de entrega y los tiempos disponibles. También hablamos del tema de la refactorización del código hacia una estructura modular, ya que pese a haberlo logrado, nos sigue pareciendo un tema de dificultad que requiere mucha más práctica respecto al tiempo que le pudimos dedicar en este trabajo.

Como mejora futura, sería interesante permitir al usuario tanto comparar varios algoritmos en ambas etapas (búsqueda y ordenamiento), como incorporar visualizaciones más intuitivas a nivel visual dentro de la terminal para evidenciar la diferencia de tiempos de ejecución en cada una de esas comparaciones. En síntesis, este trabajo no solo fortaleció y amplió las habilidades técnicas de los estudiantes, sino que también fomentó el análisis crítico en el marco de las decisiones de diseño del programa.

7. Bibliografía

- 4Geeks Academy. (s.f.). *Algoritmos de ordenamiento y búsqueda en Python*. 4Geeks. Recuperado el 1 de junio de 2025, de https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- DataCamp. (s.f.). *Búsqueda binaria en Python: guía para principiantes*. Recuperado el 6 de junio de 2025, de https://www.datacamp.com/es/tutorial/binary-search-python
- Python Software Foundation. (s.f.). The Python Standard Library. En Python 3 documentation. Recuperado el 1 de junio de 2025, de https://docs.python.org/3/
- Real Python. (s.f.). *Sorting Algorithms in Python: A Practical Guide*. Recuperado el 1 de junio de 2025, de https://realpython.com/sorting-algorithms-python/
- *Tecnicatura Universitaria en Programación a Distancia. Programación I. Unidad 10: Búsqueda y Ordenamiento* (2025). (1° ed.). Universidad Tecnológica Nacional.
- W3Schools. (s.f.). Python Examples. Recuperado de: https://www.w3schools.com/python/python_examples.aspW3Schools. (s.f.). *Python Examples*. Recuperado de: https://www.w3schools.com/python/python_examples.asp

8. Anexos

Enlaces a los diferentes recursos:

- Repositorio en GitHub con todos los recursos: https://github.com/santiagovOK/programacion-uno_integrador_VarelaSosa
- Enlace al código completo en GitHub: https://github.com/santiagovOK/programacion-uno_integrador_VarelaSosa/blob/main/programacion-uno_integrador.py
- Video en YouTube: https://youtu.be/F-snpfuKj84
- Presentación utilizada en el video: https://whimsical.com/presentacion-algoritmos-de-busqueda-y-ordenamiento-varelasosa-UgHSFeC8BqS2QsNA16AZZi
- Breve demostración en formato .gif del programa funcionando: https://github.com/santiagovOK/programacion-uno_integrador_VarelaSosa/blob/main/assets/demo.GIF