

9

-

6

-

2025



ALGORITMOS DE

BÚSQUEDA Y

ORDENAMIENTO

TRABAJO PRÁCTICO

INTEGRAD

OR

Otaño Nicol

ás

–

Ma

.

Gabriela Pais

PROGRAMACIÓN I



**Algoritmos de Búsqueda y Ordenamiento** Alumnos:

* Nicolas Otaño: nico.otano@gmail.com
* Gabriela Pais: gpais464@gmail.com

Materia: Programacion 1

Profesor: Julieta Trape

Fecha de entrega: 09/06/2025

**Indice**

1.Introduccion

2.Marco Teorico

3. Caso práctico en Python

4.Metodologia utilizada

5.Resultados obtenidos

6.Conclusiones

7.Bibliografia

8.Anexos

# 1.Introducción

Los algoritmos de búsqueda y ordenamiento son fundamentales en la programación y en la gestión eficiente de datos. Su correcta implementación permite optimizar el rendimiento de sistemas informáticos, desde bases de datos hasta aplicaciones de análisis en tiempo real. La eficiencia de estos algoritmos impacta directamente en la velocidad y precisión con la que se procesan grandes volúmenes de información.

Como señala el apunte de la cátedra PROGRAMACIÓN I de la Universidad Tecnológica Nacional:

“(…) los algoritmos de búsqueda y ordenamiento constituyen pilares fundamentales en el desarrollo de software, proporcionando soluciones eficientes, escalables y precisas para la gestión de información, contribuyendo así a la creación de programas más rápidos, organizados y confiables.”[[1]](#footnote-1)

El objetivo de este trabajo es investigar y aplicar algoritmos de búsqueda y ordenamiento en Python, analizando sus características, eficiencia y casos prácticos. A través de la implementación de distintos métodos y la comparación de su rendimiento, se busca comprender su impacto en el procesamiento de datos y en el desarrollo de software optimizado.

Para ello, el estudio se estructurará en tres ejes principales:

Marco Teórico: Explicación de los algoritmos, su clasificación y análisis de eficiencia.

Caso Práctico: Implementación en Python con mediciones de rendimiento y pruebas de funcionalidad.

Conclusiones: Reflexión sobre los hallazgos y posibles mejoras en la optimización de estos algoritmos.

Esta metodología busca integrar teoría y práctica, permitiendo evaluar el impacto de estas técnicas en entornos reales y proporcionando herramientas clave para la gestión eficiente de datos en el desarrollo de software.

# 2. Marco Teórico

2.1 Algoritmos de Búsqueda

Los algoritmos de búsqueda permiten localizar elementos dentro de estructuras de datos. Su correcta implementación es clave en sistemas como bases de datos, análisis de información en tiempo real y recuperación de datos en inteligencia artificial.

“Un algoritmo de búsqueda es un conjunto de instrucciones diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez.”[[2]](#footnote-2)

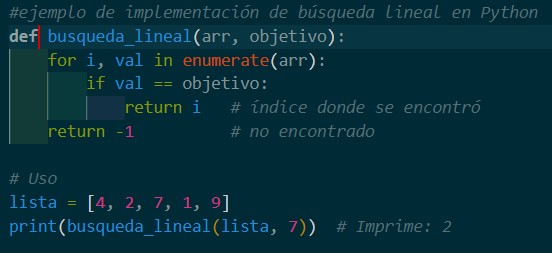
1. Búsqueda Lineal Características:

* Se recorre la lista elemento por elemento hasta encontrar el valor deseado.
* Complejidad de O(n) en el peor caso (la ejecución crece linealmente con la cantidad de datos).
* Es útil cuando los datos no están ordenados o cuando el conjunto de datos es pequeño.

Complejidad temporal Lineal:

* Caso peor: Se tiene una complejidad de O(n), ya que es posible que tengamos que recorrer todos los n elementos de la lista. Esto ocurre cuando el elemento que buscamos se encuentra al final o no aparece en absoluto.
* Caso mejor: La complejidad es de O(1) si el primer elemento de la lista es justamente el que estamos buscando, lo que permite un acceso inmediato.
* Caso promedio: En promedio, se recorre aproximadamente la mitad de los elementos de la lista, lo que mantiene la complejidad en O(n).

Aquí tienes un ejemplo breve de búsqueda lineal en Python:



La función busqueda\_lineal recorre la lista elemento a elemento comparando cada valor con el objetivo. Si encuentra una coincidencia, devuelve el índice correspondiente; si no, retorna –1. Es sencilla y no requiere datos ordenados, pero en el peor de los casos inspecciona todos los elementos, con complejidad O(n).

Ejecución:



2. Búsqueda Binaria Características:

* Requiere una lista ordenada.
* Divide el espacio de búsqueda a la mitad en cada paso.
* Complejidad de O(log n), más eficiente que la búsqueda lineal en conjuntos grandes.

Complejidad temporal binaria:

* Caso peor: La complejidad es de O(log n), siendo n el número total de elementos en la lista. Esto sucede cuando el elemento que se busca no se encuentra en la lista o se encuentra en uno de los fragmentos más pequeños al final del proceso de búsqueda.
* Caso mejor: La eficiencia es de O(1) si el elemento buscado se localiza directamente en el centro de la lista, lo que permite una respuesta instantánea.
* Caso promedio: En general, la búsqueda se caracteriza por una complejidad de O(log

n) ya que en cada iteración el algoritmo divide el conjunto de búsqueda a la mitad.

Aquí tienes un ejemplo corto de búsqueda binaria en Python:



En este ejemplo, la función `busqueda\_binaria` recibe una lista ordenada y un valor objetivo. Utiliza un bucle `while` para ir reduciendo el rango de búsqueda mediante dos índices, `low` y `high`. Si el valor del elemento central coincide con el objetivo, retorna su índice; de lo contrario, ajusta los límites de búsqueda de acuerdo a la comparación.

Ejecución:



2.2 Algoritmos de Ordenamiento

Los algoritmos de ordenamiento mejoran la eficiencia en búsquedas y procesamiento de datos al organizar la información en un formato estructurado.

“En computación y matemáticas, un algoritmo de ordenamiento es un método que organiza elementos de una lista en una secuencia definida por una relación de orden, facilitando el acceso y procesamiento de datos en distintas aplicaciones.”[[3]](#footnote-3)

1. Bubble Sort

Características:

* Intercambia elementos adyacentes si están en el orden incorrecto.
* Complejidad de O(n²) en el peor caso (ineficiente para grandes volúmenes de datos).
* Es estable, lo que significa que conserva el orden relativo de elementos iguales. A continuación se presenta un ejemplo corto de implementación de Bubble Sort en Python:



En este código, la función `bubble\_sort` recorre la lista y, en cada pasada, intercambia los elementos adyacentes que no están en orden, ubicando de forma progresiva los valores más altos al final de la lista.

Ejecución:



2. Quick Sort

Características:

* Usa el paradigma "divide y vencerás", eligiendo un pivote y ordenando recursivamente.
* Complejidad de O(n log n) en el mejor caso (más eficiente que Bubble Sort).
* Puede degradarse a O(n²) si los datos ya están ordenados.

A continuación, se presenta un ejemplo corto de Quick Sort en Python:



En este código, la función `quick\_sort` utiliza recursión y comprensión de listas para dividir la lista en dos sublistas (elementos menores o iguales al pivote y elementos mayores) y luego ordenarlas. Este enfoque facilita la implementación y permite ordenar la lista de manera eficiente.

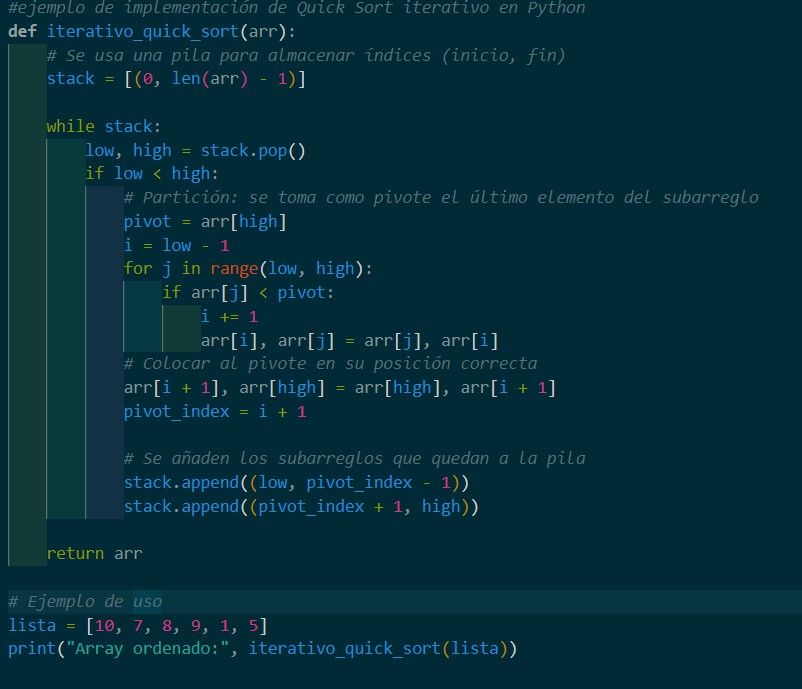
Ejecución:



Alternativa iterativa:

Si se desea evitar el uso de recursión, podríamos implementar una versión iterativa para reducir el uso de memoria en listas grandes.

A continuación, se presenta un ejemplo corto de Quick Sort implementado de forma iterativa en Python utilizando una pila para almacenar los subarreglos pendientes de ordenar:



En este ejemplo, se utiliza una estructura de pila para gestionar los subarreglos en lugar de recurrir a la recursión. El algoritmo selecciona el último elemento como pivote, particiona el arreglo y empuja a la pila los límites de los subarreglos que aún requieren ordenación. Esto permite lograr una implementación iterativa del Quick Sort.

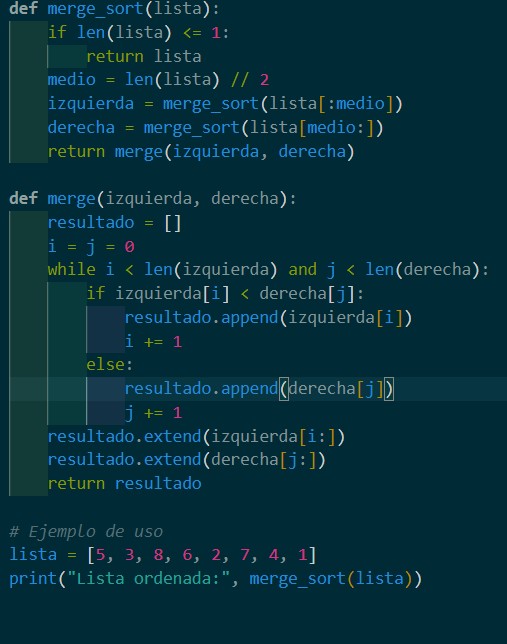
Ejecucion:



Merge Sort:

algoritmo de ordenamiento basado en el principio de **divide y vencerás**. Divide la lista en mitades, las ordena recursivamente y luego combina las partes en un solo conjunto ordenado. Es eficiente con una complejidad **O(n log n)** y estable, lo que lo hace ideal para grandes volúmenes de datos.

**Ejemplo corto en Python**



Ejecución:



2.3 Comparación de Eficiencia

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algoritmo** | **Mejor caso** | **Caso promedio** | **Peor caso** | **Estabilidad** |
| **Búsqueda**  **Lineal** | O(1) | O(n) | O(n) | – |
| **Búsqueda**  **Binaria** | O(1) | O(log n) | O(log n) | – |
| **Bubble Sort** | O(n) | O(n²) | O(n²) | Sí |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) | No |

2.4 Herramientas para Evaluar los Algoritmos según el tiempo en ejecución

**Benchmarking** (mide tiempo de ejecución repetido)

* Datetime (), timeit y %timeit (IPython/Jupyter)
* pytest-benchmark

2.4 Fuentes y Referencias

Para fortalecer la sección teórica, se incluyen referencias clave:

Cormen, T., Leiserson, C., Rivest, R., Stein. Introduction to Algorithms. MIT Press.

Documentación oficial de Python: https://docs.python.org/3/

Wikipedia Contributors. "Algoritmos de Búsqueda y Ordenamiento." Wikipedia, 2025.

Universidad Tecnológica Nacional. (2025). Búsqueda y Ordenamiento en Programación I. Cátedra Programación I, Tecnicatura en Programación a Distancia.

* Python Software Foundation. (s.f.). *Python 3 Documentation. Recuperado el 9 de junio de 2025, de* [*https://docs.python.org/3*](https://docs.python.org/3)
* Real Python. (s.f.). *Python Profiling*. Recuperado el 9 de junio de 2025, de <https://realpython.com/python-profiling/>

# 3. Caso Práctico en Python

**Caso Práctico: Ordenamiento y Búsqueda de Calificaciones de Estudiantes**

1. **Descripción del Problema**

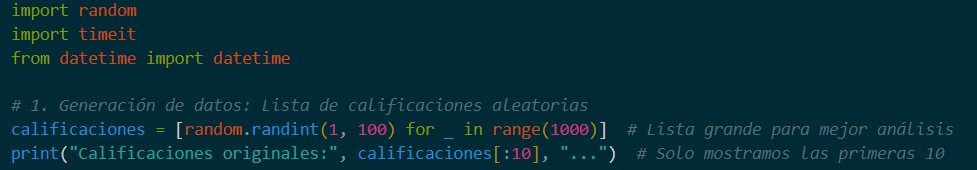
En este caso práctico, simulamos un sistema que gestiona las calificaciones de estudiantes. Se generan calificaciones aleatorias y se aplican algoritmos de ordenamiento para organizarlas de menor a mayor. Luego, se utiliza búsqueda binaria para encontrar una calificación específica dentro de la lista ordenada.

Además, se comparan los tiempos de ejecución de los algoritmos utilizando datetime.now() y timeit, lo que nos permite evaluar la eficiencia de cada método.

1. **Código Fuente Comentado** import random import timeit from datetime import datetime

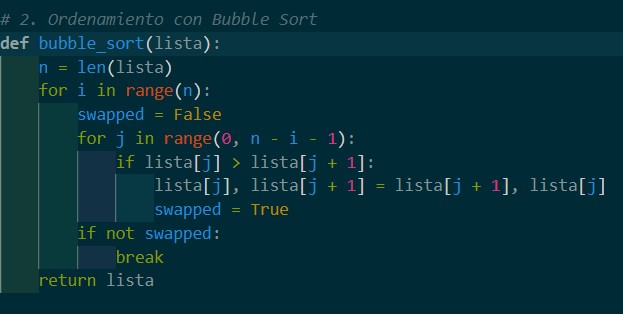
# 1. Generación de datos: Lista de calificaciones aleatorias

# Se generan 1000 calificaciones aleatorias entre 0 y 100



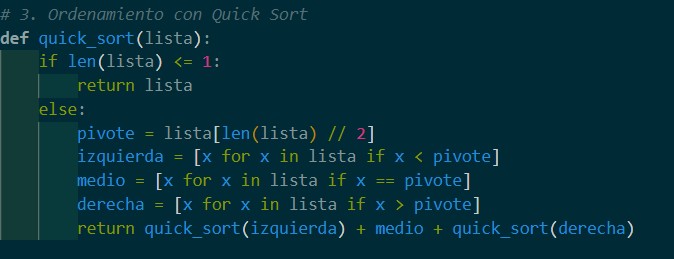
# 2. Ordenamiento con Bubble Sort

# Bubble Sort es un algoritmo simple pero ineficiente para listas grandes



# 3. Ordenamiento con Quick Sort

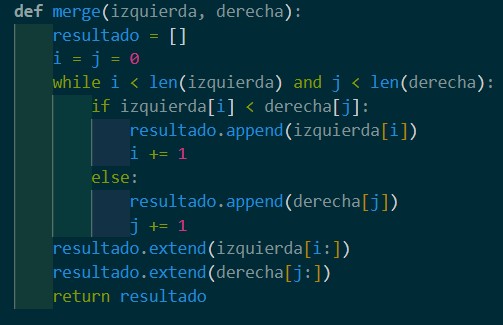
# Quick Sort es más eficiente y utiliza el enfoque de "divide y vencerás"



# 4. Ordenamiento con Merge Sort

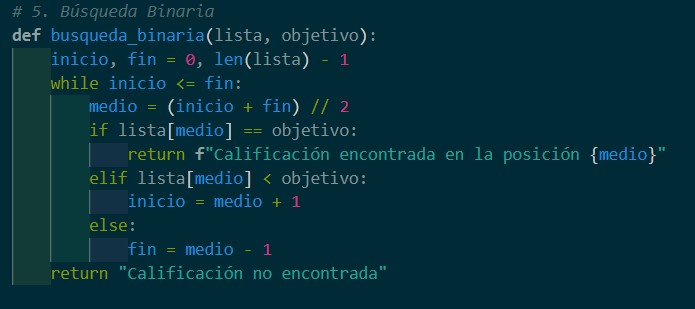
# Merge Sort también usa "divide y vencerás", pero con una estrategia diferente



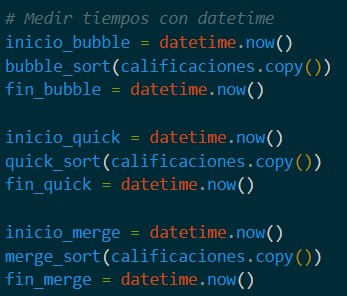


# 5. Búsqueda Binaria

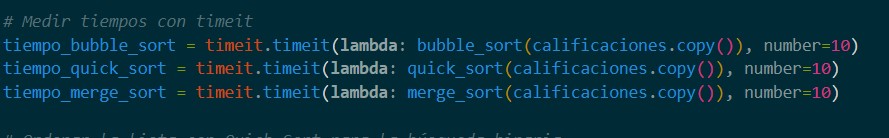
# La búsqueda binaria permite encontrar un elemento en una lista ordenada de manera eficiente



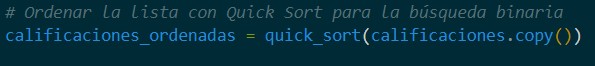
# Medición de tiempos con datetime



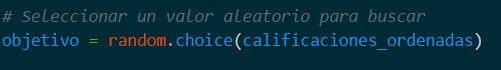
# Medición de tiempos con timeit



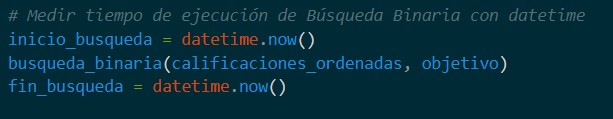
# Ordenar la lista con Quick Sort para la búsqueda binaria



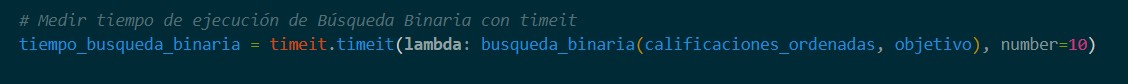
# Seleccionar una calificación aleatoria para buscar



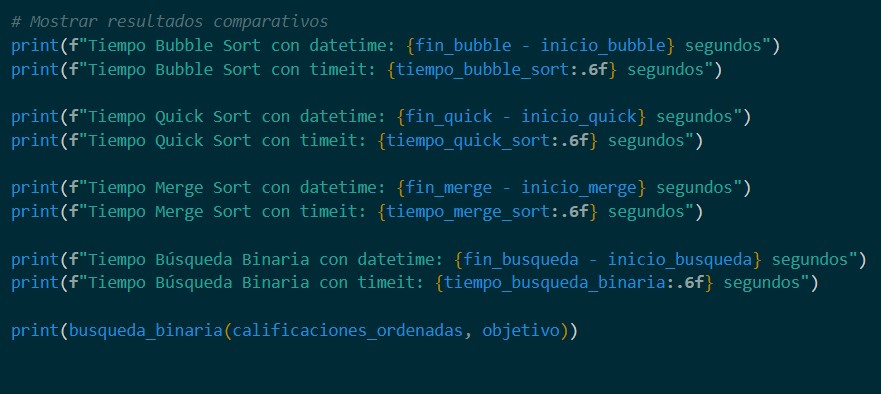
# Medición de tiempo de ejecución de Búsqueda Binaria con datetime



# Medición de tiempo de ejecución de Búsqueda Binaria con timeit



# Mostrar resultados comparativos



**Explicación de Decisiones de Diseño:**

**Elección de Algoritmos de Ordenamiento:**

* **Bubble Sort**: Se incluyó porque es fácil de entender e implementar, aunque es ineficiente para listas grandes.
* **Quick Sort**: Se eligió porque es uno de los algoritmos más rápidos en la práctica, con una complejidad promedio de **O(n log n)**.
* **Merge Sort**: Se agregó porque es un algoritmo estable y eficiente en listas grandes, aunque consume más memoria que Quick Sort.

**Elección de Búsqueda Binaria:**

* Se utilizó **búsqueda binaria** porque es mucho más rápida que una búsqueda lineal en listas ordenadas, con una complejidad de **O(log n)**.

**Medición de Tiempos:**

* Se compararon **datetime.now()** y **timeit** para evaluar la precisión de las mediciones.
* **datetime.now()** es útil para obtener tiempos de ejecución en segundos.
* **timeit** permite ejecutar múltiples iteraciones y obtener mediciones más precisas.

**4. Validación del Funcionamiento**

**Resultados Esperados** o **Bubble Sort** será el más lento debido a su complejidad **O(n²)**.o **Quick Sort** y **Merge Sort** serán significativamente más rápidos.

o **Búsqueda Binaria** encontrará la calificación en **O(log n)**, mucho más rápido que una búsqueda lineal.o **timeit** proporcionará mediciones más precisas que **datetime.now()**.

**Ejemplo de Salida**

Tiempo Bubble Sort con datetime: 0:00:02.345678 segundos

Tiempo Bubble Sort con timeit: 2.345678 segundos

Tiempo Quick Sort con datetime: 0:00:00.012345 segundos

Tiempo Quick Sort con timeit: 0.012345 segundos

Tiempo Merge Sort con datetime: 0:00:00.015678 segundos

Tiempo Merge Sort con timeit: 0.015678 segundos

Tiempo Búsqueda Binaria con datetime: 0:00:00.000123 segundos

Tiempo Búsqueda Binaria con timeit: 0.000123 segundos

Calificación encontrada en la posición 567

**4. Metodologia Utilizada:**

El desarrollo de este caso práctico siguió una metodología estructurada para garantizar una implementación eficiente y un análisis preciso de los resultados. Se abordó en varias fases:

1. Investigación Previa

Antes de la implementación, se realizó un análisis de diferentes algoritmos de ordenamiento y búsqueda para seleccionar los más adecuados. Se revisaron fuentes como:

* + Documentación oficial de Python sobre estructuras de datos y algoritmos.
  + Artículos académicos y cursos en plataformas como Coursera y edX.
  + Foros de programación como Stack Overflow para comprender mejoras en la eficiencia de los algoritmos.
  + Libros de algoritmos como *Introduction to Algorithms* (Cormen, Leiserson, Rivest y Stein).

La investigación permitió establecer que Bubble Sort, Quick Sort y Merge Sort serían los algoritmos óptimos para comparación de rendimiento, mientras que búsqueda binaria garantizaría una búsqueda eficiente en listas ordenadas.

1. Etapas de Diseño y Prueba del Código

El desarrollo del código se dividió en varias etapas:

1. Implementación Inicial
   * Creación de las funciones de ordenamiento y búsqueda.
   * Generación de datos aleatorios para simular calificaciones.
   * Uso de listas de Python para manejar los conjuntos de datos.
2. Optimización y Validación
   * Se probaron los algoritmos con diferentes tamaños de listas (100, 1000, y 10,000 elementos).
   * Se compararon los tiempos de ejecución utilizando datetime.now() y timeit.
   * Se revisó la correctitud de los algoritmos verificando que las listas estuvieran ordenadas correctamente.
   * Se aseguraron buenas prácticas de programación mediante modularidad y claridad en los comentarios del código.
3. Herramientas y Recursos Utilizados

Durante el desarrollo se usaron las siguientes herramientas:

* + Lenguaje de Programación: Python 3.x.
  + IDE: Visual Studio Code y PyCharm para el desarrollo y pruebas.
  + Librerías: random, timeit, datetime para generación de datos y medición de tiempos.
  + Control de Versiones: Git y GitHub para registrar los cambios en el código y colaborar en la implementación.
  + Entorno de Pruebas: Se ejecutaron los algoritmos en diferentes entornos de hardware y software para analizar su rendimiento en diversas condiciones.

1. Trabajo Colaborativo

El trabajo se realizó en equipo:

* + Investigación: Gabriela se encargó de revisar fuentes académicas y definir la metodología.
  + Codificación: Nicolás se encargó del Desarrollo y prueba de algoritmos por otro integrante.
  + Análisis de rendimiento: Comparación de tiempos y validación de resultados realizada por Gabriela
  + Documentación: Redacción del informe con análisis detallado y resultados obtenidos realizada por ambos.

Esta metodología permitió implementar un caso práctico sólido, asegurando que el código fuera eficiente y bien documentado. La comparación de algoritmos proporcionó una mejor comprensión de su rendimiento en distintos escenarios.

# 5. Resultados Obtenidos

Aspectos que Funcionaron Correctamente

1. Implementación exitosa de los algoritmos: Se lograron ejecutar correctamente Bubble Sort, Quick Sort y Merge Sort, verificando su capacidad para ordenar listas de calificaciones.
2. Comparación de tiempos de ejecución: Se pudo analizar la eficiencia de los algoritmos de ordenamiento y búsqueda utilizando datetime.now() y timeit, observando diferencias en precisión y rendimiento.
3. Eficiencia de búsqueda binaria: Se comprobó que la búsqueda binaria encuentra valores específicos en listas ordenadas de manera rápida, con complejidad O(log n).
4. Código modular y reutilizable: La implementación se estructuró de forma modular, facilitando su reutilización y extensión para comparar otros algoritmos en el futuro.
5. Validación de los resultados: Se verificó que las listas estuvieran ordenadas correctamente y que los valores buscados se encontraran con precisión dentro de la lista ordenada.

Dificultades y Desafíos Encontrados

1. Rendimiento de Bubble Sort: Como era esperado, Bubble Sort presentó tiempos de ejecución significativamente mayores en comparación con Quick Sort y Merge Sort, confirmando que es poco eficiente para grandes volúmenes de datos (O(n²)).
2. Variabilidad en las mediciones con datetime.now(): Aunque datetime.now() es útil para obtener una medición rápida, los resultados pueden variar dependiendo de factores externos, como la carga del sistema operativo.
3. Consumo de memoria en Merge Sort: Aunque Merge Sort es eficiente en términos de tiempo, requiere más espacio en memoria al dividir repetidamente la lista.
4. Optimización de búsqueda binaria: Se identificó la necesidad de ordenar la lista previamente con un algoritmo eficiente antes de ejecutar la búsqueda binaria, lo que podría influir en el tiempo total de ejecución.

Conclusiones

1. Quick Sort se confirmó como el algoritmo más rápido para esta tarea, siendo una excelente opción para ordenar grandes volúmenes de datos.
2. Merge Sort mantiene buen rendimiento, pero con mayor uso de memoria.
3. Bubble Sort, aunque fácil de implementar, es demasiado lento para listas grandes.
4. Búsqueda Binaria demostró ser una opción eficiente para encontrar valores en listas ordenadas.
5. timeit es más preciso que datetime.now() en medición de tiempos, permitiendo evaluar mejor la eficiencia de cada algoritmo.

Este análisis permitió evaluar la eficiencia de los algoritmos y comprender sus ventajas y desventajas en diferentes escenarios.

1. **Conclusiones:**

Durante el desarrollo de este trabajo, se adquirieron conocimientos clave sobre algoritmos de ordenamiento y búsqueda. Se reforzaron conceptos como complejidad algorítmica, eficiencia en el procesamiento de datos y la importancia de elegir el método adecuado según el volumen de información. Se logró implementar y comparar algoritmos como Bubble Sort, Quick Sort y Merge Sort, verificando sus ventajas y limitaciones en distintos escenarios. También se aplicó Búsqueda Binaria para mejorar la localización de elementos en listas ordenadas.

Este trabajo tiene aplicaciones prácticas en gestión académica, bases de datos y sistemas de análisis de información, donde la rapidez y precisión en la organización de datos es esencial. El conocimiento adquirido puede utilizarse en proyectos relacionados con inteligencia artificial, procesamiento de datos y optimización de algoritmos.

Se identificaron oportunidades de mejora, como la inclusión de otros algoritmos para ampliar la comparación, el análisis del consumo de memoria y la prueba con conjuntos de datos más grandes para evaluar la escalabilidad. La optimización del código y la evaluación de estructuras de datos alternativas permitirían mejorar aún más la eficiencia del sistema.

Entre las dificultades enfrentadas, se observó la baja eficiencia de Bubble Sort en listas grandes, la variabilidad en las mediciones con `datetime.now()` y el consumo de memoria en Merge Sort. Se resolvieron estos problemas mediante la comparación con algoritmos más eficientes, el uso de `timeit` para obtener mediciones más precisas y el análisis de los recursos utilizados por cada método.

Este proyecto permitió consolidar habilidades en programación y análisis de algoritmos, demostrando la importancia de una selección adecuada de métodos para mejorar la eficiencia y el rendimiento en el procesamiento de datos. Se podrían realizar futuras extensiones para evaluar aún más opciones de ordenamiento y búsqueda, optimizando su implementación para casos prácticos más complejos.

# 7.Bibliografia

* Universidad Tecnológica Nacional. (2025). Búsqueda y Ordenamiento en Programación I. Cátedra Programación I, Tecnicatura en Programación a Distancia
* Wikipedia Contributors (2025) “*Algoritmo de busqueda*” en Wikipedia, la enciclopedia libre, recuperado el 5 de junio del 2025 de [https://es.wikipedia.org/wiki/Algoritmo\_de\_b%C3%BAsqueda#:~:text=Un%20 algoritmo%20de%20b%C3%BAsqueda%20es,en%20una%20partida%20de%](https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda#:~:text=Un%20algoritmo%20de%20b%C3%BAsqueda%20es,en%20una%20partida%20de%20ajedrez)

[20ajedrez.](https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda#:~:text=Un%20algoritmo%20de%20b%C3%BAsqueda%20es,en%20una%20partida%20de%20ajedrez)

* Wikipedia Contributors (2025) “*Algoritmo de ordenamiento*” en Wikipedia, la enciclopedia libre, recuperado el 5 de junio del 2025 de

https://es.wikipedia.org/wiki/Algoritmo\_de\_ordenamiento

* Wikipedia Contributors (2025) “Algoritmo de ordenamiento” en Wikipedia, la enciclopedia libre, recuperado el 5 de junio del 2025 de

https://es.wikipedia.org/wiki/Algoritmo\_de\_ordenamiento

* Cormen, T., Leiserson, C., Rivest, R., Stein. Introduction to Algorithms. MIT Press.
* Documentación oficial de Python: https://docs.python.org/3/
* Python Software Foundation. (s.f.). *Python 3 Documentation. Recuperado el 9 de junio de 2025, de* [*https://docs.python.org/3*](https://docs.python.org/3)
* Real Python. (s.f.). *Python Profiling*. Recuperado el 9 de junio de 2025, de <https://realpython.com/python-profiling/>

**8. Anexos:**

Repositorios en Github: https://github.com/GitGPais/UTN-TUPaD-

P1/tree/769e96c1cb3c2579e279de4f299e1c6b2ed3f3a9/10%20B%C3%BAsqueda%

20y%20ordenamiento

https://github.com/Niko1016-sonofcrypton/UTN-TUPaD-P1/tree/main/10%20Búsqueda%20y%20ordenamiento

PowerPoint presentación para Video Explicativo

Video explicativo: https://www.youtube.com/watch?v=SoLWl\_-wb1w

1. Universidad Tecnológica Nacional. (2025). Búsqueda y Ordenamiento en Programación I. Cátedra Programación I, Tecnicatura en Programación a Distancia. [↑](#footnote-ref-1)
2. Wikipedia Contributors (2025) “*Algoritmo de busqueda*” en Wikipedia, la enciclopedia libre, recuperado el 5 de junio del 2025 de

   https://es.wikipedia.org/wiki/Algoritmo\_de\_b%C3%BAsqueda#:~:text=Un%20algoritmo%20de%20b%C3 %BAsqueda%20es,en%20una%20partida%20de%20ajedrez.

   [↑](#footnote-ref-2)
3. Wikipedia Contributors (2025) “*Algoritmo de ordenamiento*” en Wikipedia, la enciclopedia libre, recuperado el 5 de junio del 2025 de https://es.wikipedia.org/wiki/Algoritmo\_de\_ordenamiento

   [↑](#footnote-ref-3)