Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Trabajo Integrador N°1

Búsqueda y ordenamiento

Programación I

Docente Titular

Julieta Trapé

Docente Tutor

Sofía Raia

Estudiantes

Miguel Pichulman

Rodriguez Joaquin

9 de junio de 2025

índice

[Introducción 3](#_Toc200398219)

[Marco Teórico 4](#_Toc200398220)

[Caso práctico 8](#_Toc200398221)

[Metodología Utilizada 11](#_Toc200398222)

[Resultados Obtenidos 12](#_Toc200398223)

[Conclusiones 13](#_Toc200398224)

[Bibliografía 14](#_Toc200398225)

[Anexos 15](#_Toc200398226)

# Introducción

Los algoritmos de búsqueda y ordenamiento cumplen un rol fundamental para el manejo eficiente de datos. A medida que las aplicaciones procesan grandes volúmenes de información, es fundamental contar con técnicas que faciliten la localización y organización eficiente de los datos. Los algoritmos de ordenamiento permiten reestructurar estructuras de datos siguiendo un criterio determinado, como el orden numérico o alfabético. Por su parte, los algoritmos de búsqueda se utilizan para encontrar elementos específicos dentro de esas estructuras de datos, y su rendimiento puede variar significativamente dependiendo de si los datos estén previamente ordenados.

Este trabajo tiene como objetivo explorar distintos métodos de ordenamiento y búsqueda, con especial énfasis en la búsqueda binaria, dada su eficiencia en estructuras de datos ordenadas. Se implementarán estos algoritmos en Python, analizando su funcionamiento, ventajas y limitaciones.

# Marco Teórico

**Algoritmo.**

Un algoritmo es un conjunto de instrucciones diseñado para resolver un problema o realizar una tarea. Los algoritmos se escriben generalmente de una manera paso a paso para que puedan ser fácilmente seguidos por una computadora.

Para que un algoritmo sea efectivo, debe ser capaz de recibir datos de entrada, procesar esos datos y producir una salida. Los algoritmos pueden ser diseñados para realizar una amplia variedad de tareas, como ordenar una lista de números o encontrar un elemento particular en una base de datos.

La eficiencia de un algoritmo se mide a menudo por su tiempo de ejecución, que es la cantidad de tiempo que tarda el algoritmo en completar su tarea. El tiempo de ejecución de un algoritmo puede verse afectado por el tipo de datos de entrada que se procesan, así como por el número de elementos que deben procesarse.

Los algoritmos de búsqueda y ordenamiento son herramientas clave para organizar y acceder a los datos de manera eficiente a medida que las aplicaciones manejan grandes volúmenes de información. La correcta organización y localización de los datos resulta esencial para optimizar el rendimiento y garantizar respuestas rápidas y precisas.

**Algoritmos de Ordenamiento**

Según Thomas H. Cormen(2009), un algoritmo de ordenamiento es un procedimiento que reorganiza un conjunto de elementos en un orden específico, generalmente ascendente o descendente, optimizando el acceso y procesamiento de los datos.

**Tipos de Algoritmos de Ordenamiento**

Los algoritmos de ordenamiento son fundamentales en la informática y juegan un papel clave en la organización eficiente de datos. Su correcto uso permite optimizar tiempos de procesamiento y mejorar el acceso a la información. Tienen aplicación en múltiples áreas de la informática:

* Motores de Búsqueda: estos ordenan las páginas web según su relevancia y otros factores.
* Bases de Datos: se utilizan para organizar registros y mejorar la eficiencia de las consultas.
* E-commerce: se utilizan para ordenar los productos por precio, popularidad o relevancia

Existen diversos métodos de ordenamiento, cada uno con ventajas y desventajas según la estructura y cantidad de datos a procesar. Algunos se basan en comparaciones entre elementos, mientras que otros aprovechan estructuras especializadas para mejorar la velocidad del ordenamiento.

Algunos de los más utilizados son:

**Burbuja (Bubble Sort)**

El algoritmo de ordenamiento por burbuja es uno de los más simples, pero menos eficientes. Recorre repetidamente una lista, compara elementos adyacentes y los intercambia si están en el orden incorrecto, este procedimiento se repite hasta que no se requieren más intercambios. Por ejemplo, imagina que tenes una fila de jugadores de fútbol con los números desordenados en sus camisetas. Cada uno compara su número con el de al lado y, si está en el orden incorrecto, intercambian lugares, y repite este proceso varias veces hasta que todos estén ordenados.

Ventajas:

* fácil de entender e implementar
* útil para listas pequeñas

Desventajas:

* ineficiente para grandes volúmenes de datos
* realiza muchas comparaciones innecesarias

**Por Inserción (Insertion Sort)**

El algoritmo de ordenamiento por inserción es un algoritmo simple pero eficiente. Funciona dividiendo la lista en dos partes, una parte ordenada y otra desordenada, a medida que se recorre la lista desordenada, se insertan elementos en la posición correcta en la parte ordenada. Cada elemento se inserta en la posición correcta respecto a los elementos ya ordenados, simulando cómo ordenaríamos cartas en la mano. Por ejemplo, ordenar facturas o papeles por fecha. Vas agarrando uno por uno de una pila desordenada y los insertas en una carpeta, asegurándote de colocarlos en el lugar correcto según la fecha, a medida que sumás más documentos, la carpeta siempre se mantiene ordenada.

Ventajas:

* su implementación es simple

Desventajas:

* Poco eficiente para listas grandes (complejidad O(n²)).

**Por Selección (selection sort)**

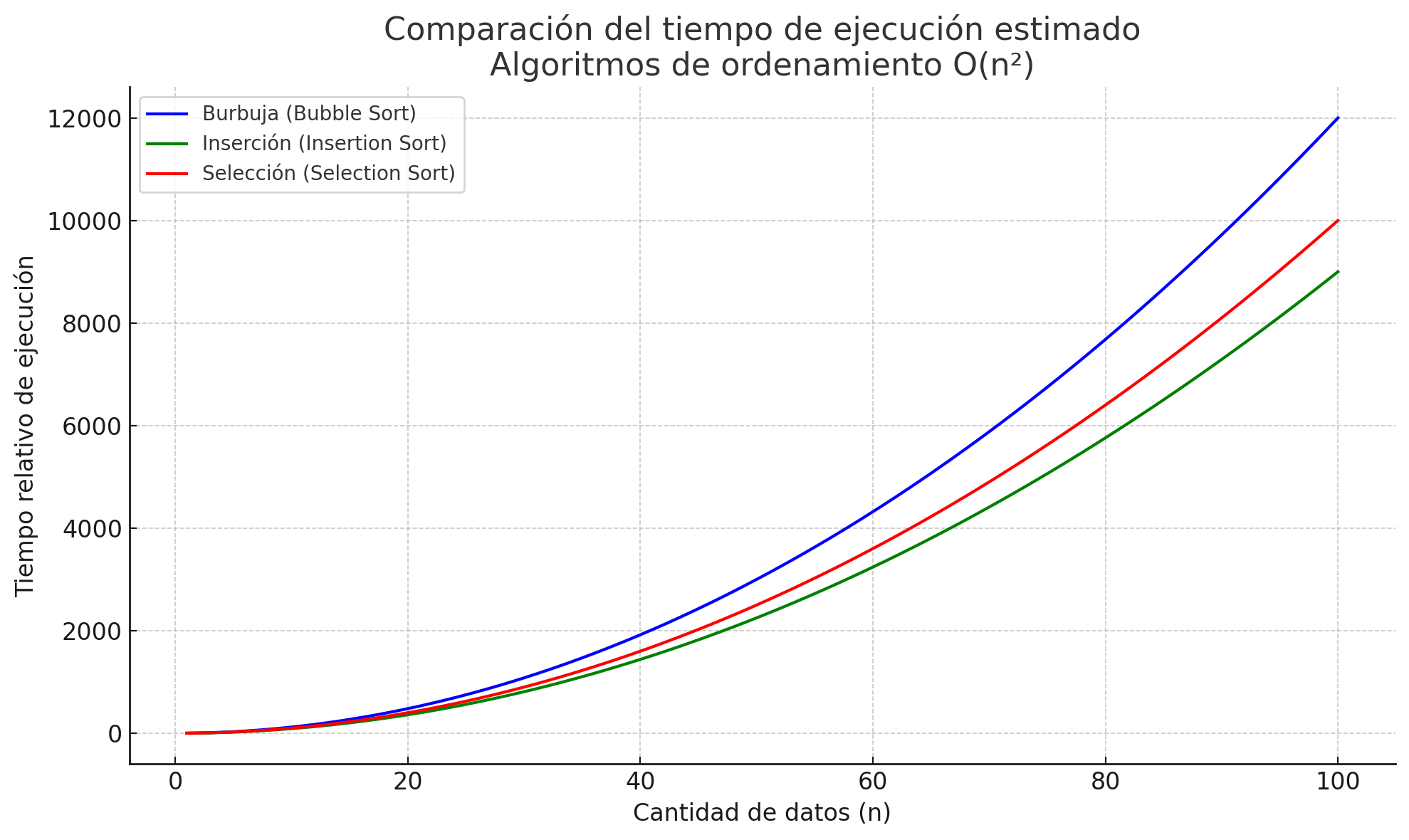
Consiste en encontrar el menor de los elementos e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenarlo todo. Por ejemplo, cuando ordenas los billetes. Buscás el de menor valor, lo ponés primero; después el siguiente más chico, y así hasta tenerlos todos ordenados.

Ventajas:

* fácil de implementar
* útil para listas pequeñas

Desventajas:

* Poco eficiente para listas grandes (complejidad O(n²)).
* no es estable



**Algoritmos de Búsqueda**

Los algoritmos de búsqueda son métodos diseñados para localizar un elemento específico dentro de una lista de datos, la elección del método depende de la estructura de la lista. Por ejemplo, si los elementos están ordenados, se puede emplear la búsqueda binaria, que es eficiente. Sin embargo, si la lista está desordenada, la búsqueda binaria no es aplicable, y en su lugar se utiliza la búsqueda lineal, que recorre los elementos uno por uno hasta encontrar el objetivo.

* Búsqueda Lineal: Consiste en examinar una lista de elementos de forma secuencial, uno a uno, hasta localizar el elemento deseado. Este método es simple de programar, pero puede resultar ineficiente, especialmente en listas extensas o cuando el elemento se encuentra hacia el final.

Ventajas:

1. Fácil de entender e implementar, requiere poco código.
2. Funciona en listas desordenadas o con cualquier tipo de datos.

Desventajas:

1. Su tiempo de ejecución es O(n), lo que lo hace lento para listas grandes, ya que revisa cada elemento uno por uno.

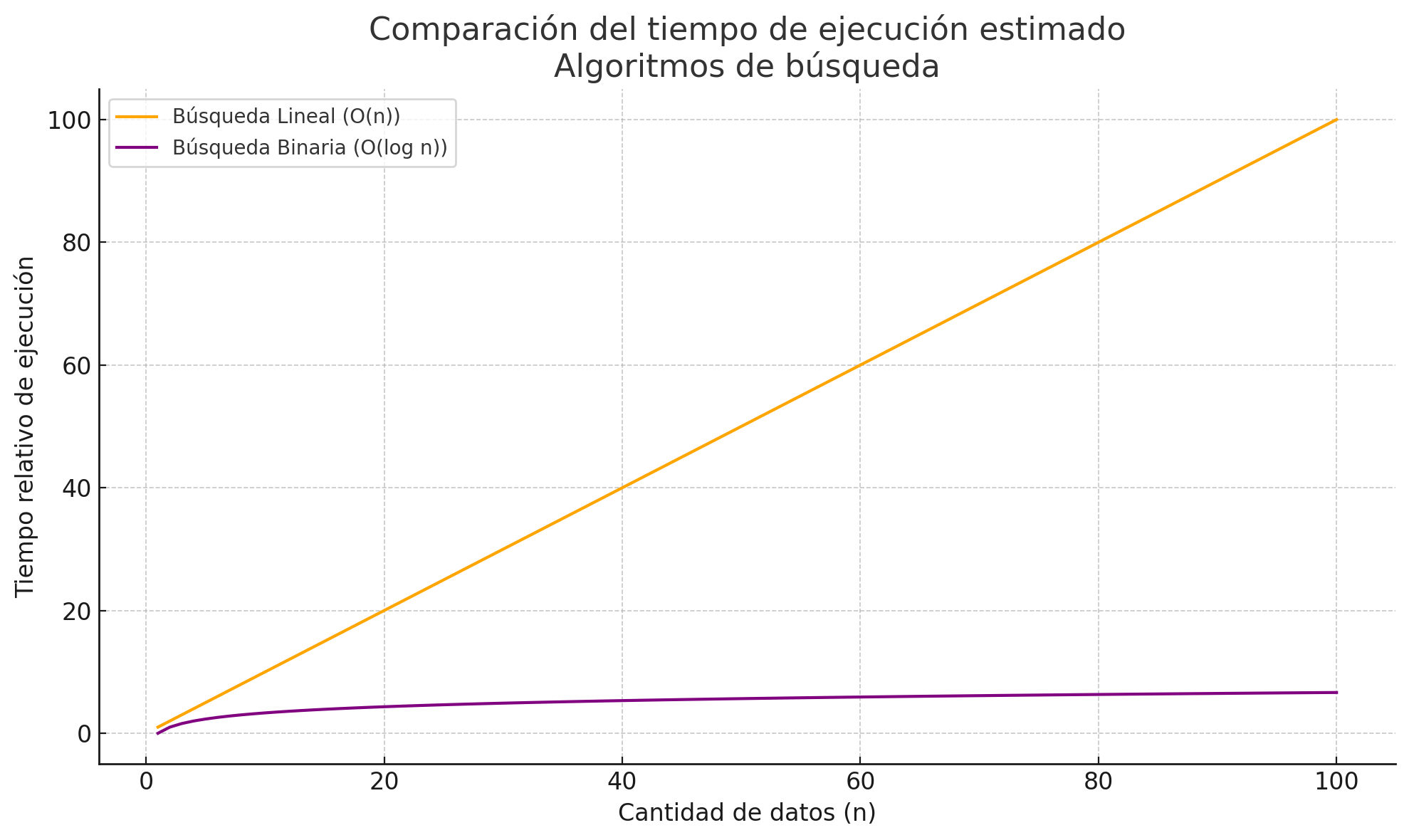
* Búsqueda Binaria: Es un método eficiente que se utiliza únicamente en listas ordenadas. Opera dividiendo la lista en dos partes de manera repetitiva y comparando el elemento objetivo con el elemento central, lo que reduce drásticamente el número de la cantidad de comparaciones necesarias.

Ventajas:

1. Tiene un tiempo de ejecución de O(log n), lo que lo hace mucho más rápido que la búsqueda lineal para listas grandes.
2. Al dividir la lista en mitades repetidamente, reduce significativamente el número de elementos a evaluar.

Desventajas:

1. Solo funciona en listas previamente ordenadas, lo que implica un costo adicional si la lista no está ordenada.
2. Su implementación es más complicada que la búsqueda lineal, especialmente si se debe considerar el ordenamiento previo.



# Caso práctico

Gestión de inventario en una tienda

Tenemos una tienda que vende productos identificados por códigos numéricos. Para optimizar la búsqueda de productos, primero ordenaremos el inventario con el algoritmo de ordenamiento por burbuja y luego utilizaremos búsqueda binaria para encontrar un producto específico.

# Importar modulo random para generar numeros aleatorios y time para medir el tiempo de la ejecucion

import random

import time

#funciones

# Por burbuja

def burbuja(lista):

n = len(lista)

for i in range(n-1):

for j in range(n-1-i):

if lista[j] > lista[j+1]:

lista[j], lista[j+1] = lista[j+1], lista[j]

# Busqueda lineal

def busqueda\_lineal(lista, elemento):

inicio = time.time() # Iniciar medicion del tiempo

for i in range(len(lista)):

if lista[i] == elemento:

fin\_tiempo = time.time()

tiempo = (fin\_tiempo - inicio) \* 1000 # Tiempo en milisegundos

return i, tiempo

fin\_tiempo = time.time()

tiempo = (fin\_tiempo - inicio) \* 1000

return -1, tiempo

# Busqueda binaria

def busqueda\_binaria(lista, elemento):

inicio = time.time() # Iniciar medicion del tiempo

indice = 0

fin = len(lista) - 1

while indice <= fin:

medio = (indice + fin) // 2

if lista[medio] == elemento:

fin\_tiempo = time.time()

tiempo = (fin\_tiempo - inicio) \* 1000 # Tiempo en milisegundos

return medio, tiempo

elif lista[medio] < elemento:

indice = medio + 1

else:

fin = medio - 1

fin\_tiempo = time.time()

tiempo = (fin\_tiempo - inicio) \* 1000

return -1, tiempo

#PROGRAMA PRINCIPAL

# Lista de productos (codigos numericos generados aleatoriamente por random)

inventario = [random.randint(100, 999) for i in range(100)]

print("Inventario antes de ordenar:", inventario)

burbuja(inventario)

print("Inventario ordenado:", inventario)

# Menu

while True:

print("\n=== Búsqueda de Productos ===")

producto\_a\_buscar = int(input("Ingrese el código del producto a buscar (0 para salir): "))

if producto\_a\_buscar == 0:

print("Saliendo del programa...")

break

# Ejecutar busqueda binaria

indice\_binaria, tiempo\_binaria = busqueda\_binaria(inventario, producto\_a\_buscar)

if indice\_binaria != -1:

print(f"El producto con código {producto\_a\_buscar} está en la posición {indice\_binaria} y se demoró {tiempo\_binaria:.5f} ms usando búsqueda Binaria.")

else:

print(f"El producto con código {producto\_a\_buscar} no fue encontrado y se demoró {tiempo\_binaria:.5f} ms usando búsqueda Binaria.")

# Ejecutar busqueda lineal

indice\_lineal, tiempo\_lineal = busqueda\_lineal(inventario, producto\_a\_buscar)

if indice\_lineal != -1:

print(f"El producto con código {producto\_a\_buscar} está en la posición {indice\_lineal} y se demoró {tiempo\_lineal:.5f} ms usando búsqueda Lineal.")

else:

print(f"El producto con código {producto\_a\_buscar} no fue encontrado y se demoró {tiempo\_lineal:.5f} ms usando búsqueda Lineal.")

# Metodología Utilizada

* Recolección de información teórica sobre algoritmos de búsqueda y ordenamiento en fuentes confiables.
* Implementación en Python de los algoritmos: burbuja, búsqueda binaria y búsqueda lineal.
* Generación de listas de datos numéricos aleatorios(random) simulando un inventario de artículos.
* Medición de tiempos de ejecución con el módulo time para comparar el rendimiento de cada algoritmo.
* Análisis de resultados y elaboración del informe final con conclusiones y anexos.

# Resultados Obtenidos

* El programa ordeno correctamente la lista de códigos generados aleatoriamente.
* La búsqueda binaria localizó de forma eficiente el código ingresado, siempre que la lista estuviera ordenada.
* La búsqueda lineal también funcionó correctamente, aunque con mayor tiempo de ejecución en comparación.
* Se comprendieron las diferencias en complejidad y eficiencia entre los algoritmos utilizados.
* Se valoró la importancia de tener los datos previamente ordenados para aplicar búsqueda binaria con éxito.

# Conclusiones

Los algoritmos de búsqueda y ordenamiento representan herramientas fundamentales para el tratamiento eficiente de datos. A través de este trabajo se pudo comprobar cómo la elección del algoritmo adecuado influye directamente en el rendimiento del sistema. La búsqueda binaria demostró ser más eficiente que la búsqueda lineal en listas largas, ya que demora menos tiempo al reducir la cantidad de comparaciones necesarias, siempre y cuando los datos estén previamente ordenados. Sin embargo, en listas cortas, la búsqueda lineal puede ofrecer un mejor rendimiento debido a su menor sobrecarga. Esto remarca la importancia de considerar tanto el tamaño de los datos como su estructura al seleccionar un algoritmo. Además, se afianzaron conceptos clave sobre complejidad algorítmica y se reforzó la necesidad de comprender el funcionamiento interno de cada técnica para aplicarlas correctamente según el contexto.

# Bibliografía

TechEdu. (s.f.). Algoritmo. En TechLib TechEdu.

<https://techlib.net/techedu/algoritmo/>

PrepInsta. (2023, 7 de octubre). Linear search in Python. <https://prepinsta.com/data-structures-and-algorithms-in-python/linear-search-in-python/>

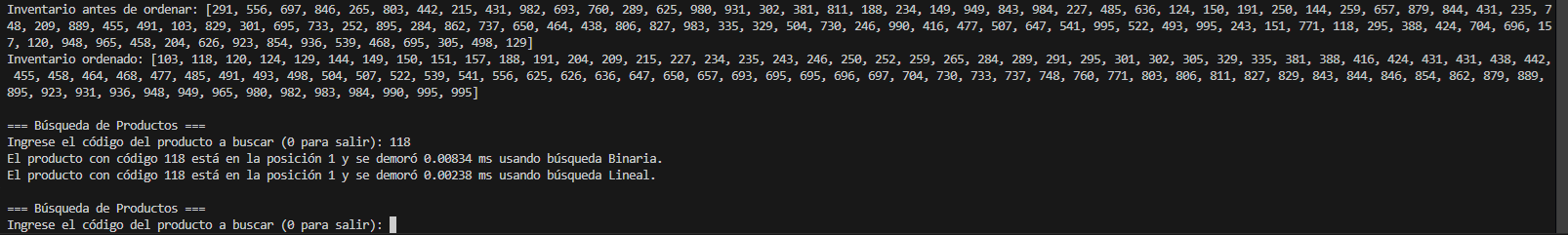
4Geeks Academy. Algoritmos de ordenamiento y búsqueda en Python. <https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python?utm_source=chatgpt.com>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3.ª ed.). The MIT Press.

# Anexos



* Se aumentó la cantidad de elementos en la lista de 100 a 500, permitiendo medir mejor los tiempos de ejecución y evidenciar cómo la búsqueda binaria se vuelve más eficiente en listas grandes, mientras que la búsqueda lineal funciona mejor en listas pequeñas.



* Se realizó la captura de la consola de visual code mostrando que el código funciona correctamente