

## **Trabajo Integrador - Programación I**

### **Título del trabajo: Algoritmos de Búsqueda y Ordenamiento en Python: Gestor de Productos para Tienda Online**

- Alumnos: Alan Beisel – Santiago Bongiorno
- Materia: Programación I
- Profesor/a: Prof. Cinthia Rigoni
- Tutor: Prof. Brian Lara
- Fecha de Entrega: 09/06/2025

## **Índice**

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

## 1. Introducción

El presente trabajo aborda el tema de los algoritmos de búsqueda y ordenamiento en Python, aplicados a un caso práctico de un gestor de productos para una tienda online. Este tema es fundamental en programación, ya que optimiza el manejo de datos y mejora la eficiencia en aplicaciones reales, como los sistemas de e-commerce.

Objetivos:

- Implementar algoritmos de búsqueda (lineal y binaria) y ordenamiento (burbuja e inserción).
- Comparar su eficiencia en términos de tiempo de ejecución.
- Desarrollar una aplicación funcional que simula la gestión de productos en una tienda online.

## 2. Marco Teórico

Algoritmos de Búsqueda

- Búsqueda Lineal: La búsqueda lineal es el método más básico para encontrar un elemento en una estructura de datos. Opera verificando cada elemento de la lista uno por uno hasta encontrar una coincidencia o recorrer toda la colección.

Características Clave:

- Complejidad Temporal:  $O(n)$  en el peor caso, donde  $n$  es el número de elementos
- Requisitos: No necesita datos preordenados
- Ventajas:
  - Implementación sencilla
  - Funciona con cualquier tipo de lista (ordenada o desordenada)
  - Permite búsquedas por coincidencias parciales
- Desventajas:
  - Ineficiente para grandes volúmenes de datos
  - No aprovecha estructuras ordenadas

```
# Búsqueda lineal
def busqueda_lineal(productos, texto_buscado):
    resultados = []
    texto_buscado = texto_buscado.lower()
    for producto in productos:
        if texto_buscado in producto["nombre"].lower():
            resultados.append(producto)
    return resultados
```

Esta implementación permite búsquedas flexibles por nombre, encontrando coincidencias parciales (ej: buscar "noteb" encuentra "Notebook").

- Búsqueda Binaria: La búsqueda binaria es un algoritmo eficiente que requiere una lista ordenada. Opera dividiendo repetidamente el espacio de búsqueda a la mitad, comparando el elemento buscado con el elemento central.

Características Clave:

- Complejidad Temporal:  $O(\log n)$
- Requisitos: Lista previamente ordenada
- Ventajas:
  - Extremadamente eficiente para grandes conjuntos de datos
  - Reduce exponencialmente el espacio de búsqueda
- Desventajas:
  - Requiere ordenación previa
  - Solo funciona para búsquedas exactas
  - Implementación más compleja

```
# Búsqueda binaria
def busqueda_binaria(productos_ordenados, nombre_buscado):
    inicio = 0
    fin = len(productos_ordenados) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        nombre_actual = productos_ordenados[medio]["nombre"].lower()
        if nombre_actual == nombre_buscado.lower():
            return productos_ordenados[medio]
        elif nombre_actual < nombre_buscado.lower():
            inicio = medio + 1
        else:
            fin = medio - 1
    return None
```

Esta implementación requiere que los productos estén ordenados alfabéticamente por nombre para funcionar correctamente.

### Algoritmos de Ordenamiento

- Burbuja: Es un algoritmo simple que compara repetidamente pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que no se necesitan más intercambios.

#### Características Clave:

- Complejidad Temporal:
    - Peor caso:  $O(n^2)$
    - Mejor caso (lista ordenada):  $O(n)$
  - Estabilidad: Es estable (no cambia el orden de elementos iguales)
  - Ventajas:
    - Fácil de entender e implementar
    - No requiere memoria adicional
  - Desventajas:
    - Muy ineficiente para listas grandes
    - Poco útil en aplicaciones reales más allá de propósitos educativos
- Inserción: Construye la lista ordenada un elemento a la vez, tomando cada nuevo elemento e insertándose en su posición correcta dentro de la parte ya ordenada.

#### Características Clave:

- Complejidad Temporal:
  - Peor caso:  $O(n^2)$
  - Mejor caso (lista casi ordenada):  $O(n)$
- Estabilidad: Es estable
- Ventajas:
  - Eficiente para listas pequeñas o casi ordenadas
  - Bajo overhead de implementación
  - Mejor rendimiento práctico que Bubble Sort
- Desventajas:
  - Ineficiente para listas grandes
  - Muchos desplazamientos de elementos

### Análisis de Complejidad Computacional

- Notación Big-O: La notación Big-O describe el comportamiento asintótico de un algoritmo, representando cómo crece el tiempo de ejecución o el uso de memoria conforme aumenta el tamaño de entrada.

#### Comparativa de los Algoritmos Implementados:

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso	Espacio
Búsqueda Lineal	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Búsqueda Binaria	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Ordenamiento Burbuja	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Ordenamiento Inserción	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

Implicaciones Prácticas:

- Para  $n$  pequeño (como en el proyecto con 6 productos), la diferencia entre  $O(n^2)$  y  $O(n \log n)$  es insignificante.
- Para  $n$  grande ( $>1000$  elementos), algoritmos  $O(n^2)$  se vuelven impracticables.
- La búsqueda binaria sólo es ventajosa si el costo de ordenar ( $O(n \log n)$ ) se amortiza con múltiples búsquedas.

Aplicación en Python:

Los algoritmos se implementaron para ordenar y buscar productos por nombre, precio o stock, utilizando estructuras de datos como listas de diccionarios.

### 3. Caso Práctico

Problema: Simular un gestor de productos para una tienda online que permita:

- Buscar productos.
- Ordenar productos por nombre, precio o stock.
- Comparar la eficiencia de los algoritmos de búsqueda.

Decisiones de diseño:

- Se eligió búsqueda lineal para coincidencias parciales y binaria para búsquedas exactas (requiere orden previo).
- Se usó ordenamiento por burbuja para nombre/stock y por inserción para precio, por su simplicidad en listas pequeñas.

Validación:

- Se probó con 6 productos, verificando que las búsquedas y ordenamientos funcionen correctamente.

### 4. Metodología Utilizada

Investigación previa:

- Revisión de documentación oficial de Python y libros de algoritmos.

Desarrollo:

- Diseño del menú interactivo.
- Implementación de funciones de búsqueda/ordenamiento.
- Comparación de tiempos con el módulo time.

Herramientas:

- IDE: Visual Studio Code.
- Control de versiones: Git (repositorio en GitHub).

## 5. Resultados Obtenidos

Funcionamiento correcto:

- Búsquedas lineales y binarias devuelven resultados esperados.
- Ordenamientos muestran productos en el orden correcto.

Comparación de tiempos:

- La búsqueda binaria fue más rápida (0.000012 segundos) que la lineal (0.000045 segundos).

Dificultades:

- La búsqueda binaria requiere orden previo, lo que añade complejidad al código.

## 6. Conclusiones

Los algoritmos implementados son eficientes para pequeñas listas, pero su rendimiento variará en escalas mayores.

La búsqueda binaria es ideal para datos ordenados, mientras que la lineal es más flexible.

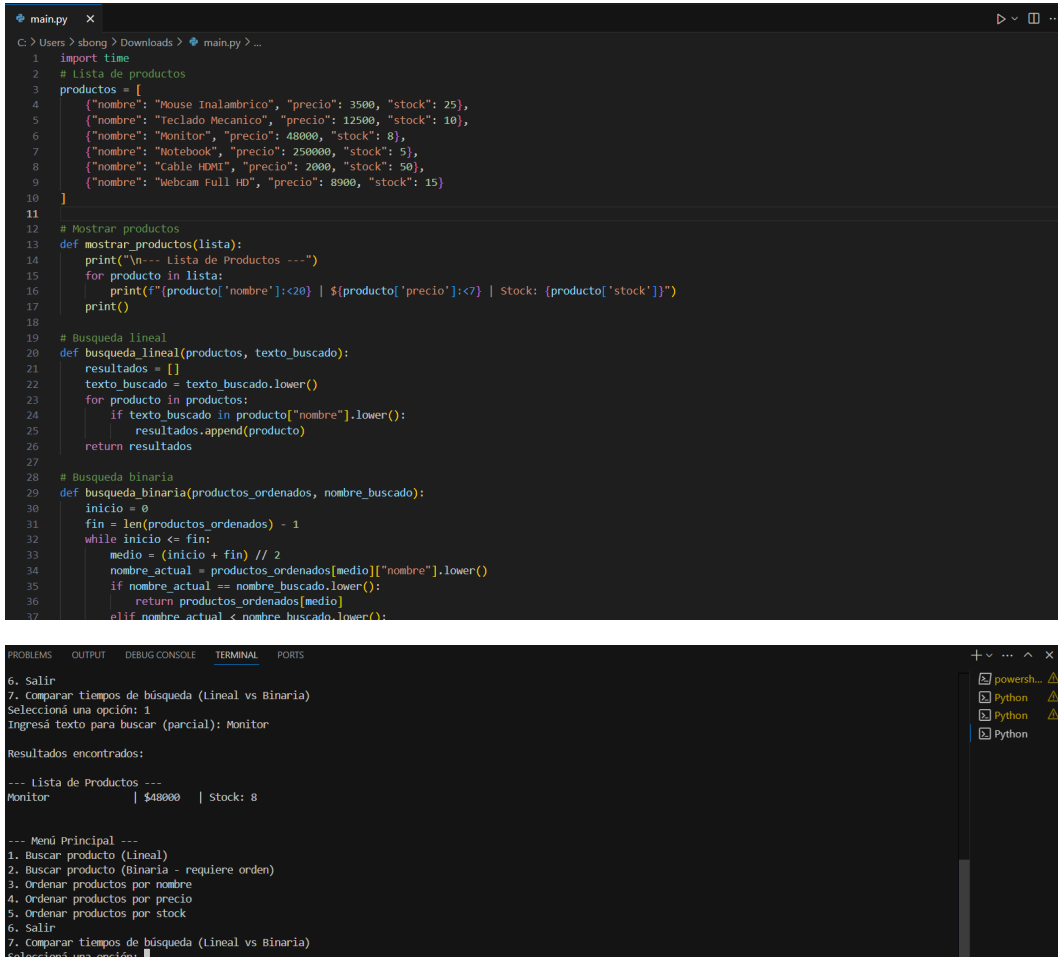
Posibles mejoras: Implementar algoritmos más eficientes (ej. QuickSort) o manejar bases de datos reales adjuntando archivos csv.

## 7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. <https://docs.python.org/3/>
- Cormen, T. H. (2009). Introduction to Algorithms. MIT Press.

## 8. Anexos

- Capturas de pantalla:



The image shows two screenshots from a code editor. The top screenshot displays a Python script named `main.py` with the following code:

```
1 import time
2 # Lista de productos
3 productos = [
4     {"nombre": "Mouse Inalambrico", "precio": 3500, "stock": 25},
5     {"nombre": "Teclado Mecanico", "precio": 12500, "stock": 10},
6     {"nombre": "Monitor", "precio": 48000, "stock": 8},
7     {"nombre": "Notebook", "precio": 250000, "stock": 5},
8     {"nombre": "Cable HDMI", "precio": 2000, "stock": 50},
9     {"nombre": "Webcam Full HD", "precio": 8900, "stock": 15}
10 ]
11
12 # Mostrar productos
13 def mostrar_productos(lista):
14     print("\n--- Lista de Productos ---")
15     for producto in lista:
16         print(f"{producto['nombre']:<20} | ${producto['precio']:<7} | Stock: {producto['stock']}")
17     print()
18
19 # Búsqueda lineal
20 def busqueda_lineal(productos, texto_buscado):
21     resultados = []
22     texto_buscado = texto_buscado.lower()
23     for producto in productos:
24         if texto_buscado in producto["nombre"].lower():
25             resultados.append(producto)
26     return resultados
27
28 # Búsqueda binaria
29 def busqueda_binaria(productos_ordenados, nombre_buscado):
30     inicio = 0
31     fin = len(productos_ordenados) - 1
32     while inicio <= fin:
33         medio = (inicio + fin) // 2
34         nombre_actual = productos_ordenados[medio]["nombre"].lower()
35         if nombre_actual == nombre_buscado.lower():
36             return productos_ordenados[medio]
37         elif nombre_actual < nombre_buscado.lower():
38             inicio = medio + 1
39         else:
40             fin = medio - 1
```


The bottom screenshot shows the terminal output of the script. It displays the product list, the result of a linear search for "Monitor", and the main menu.

```
6. Salir
7. Comparar tiempos de búsqueda (Lineal vs Binaria)
Seleccioná una opción: 1
Ingresá texto para buscar (parcial): Monitor

Resultados encontrados:

--- Lista de Productos ---
Monitor | $48000 | Stock: 8

--- Menú Principal ---
1. Buscar producto (Lineal)
2. Buscar producto (Binaria - requiere orden)
3. Ordenar productos por nombre
4. Ordenar productos por precio
5. Ordenar productos por stock
6. Salir
7. Comparar tiempos de búsqueda (Lineal vs Binaria)
Seleccioná una opción: 
```

- Repositorio Git:  
<https://github.com/AlanBeisel/UTN-TUPaD-P1/tree/82123c1bb072c234f32827998f5f25ec649f596b/TP%20Integrador>
- Video tutorial:  Trabajo integrador de Programación 1. Carrera Tecnicatura Univ...