

TRABAJO PRÁCTICO INTEGRADOR – PROGRAMACIÓN I

Título:

Optimización de la gestión hospitalaria mediante algoritmos de búsqueda y ordenamiento en Python

Alumnos:

Lautaro Lopez

Emiliano Jara

Materia: Programación I

Profesor/a: Enferrel Ariel.

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

Los algoritmos de búsqueda y ordenamiento representan herramientas esenciales en el desarrollo de software y en la manipulación eficiente de información. Su aplicación trasciende los entornos académicos, impactando directamente en procesos del mundo real, especialmente en entornos que requieren organización rápida y precisa de grandes volúmenes de datos.

En este trabajo integrador se propone explorar y aplicar estos algoritmos al contexto de un hospital, donde es vital la rápida localización y priorización de pacientes. Esta elección se justifica por su relevancia social y su aplicabilidad real, considerando que la optimización de los tiempos de atención puede incidir directamente en la calidad del servicio y la toma de decisiones médicas.

El objetivo principal es demostrar, mediante un caso práctico desarrollado en Python, cómo los algoritmos de ordenamiento y búsqueda pueden implementarse para mejorar un sistema de gestión de pacientes, permitiendo ordenar por nivel de urgencia y buscar eficientemente por número de documento.

2. Marco Teórico

2.1 Algoritmos de Ordenamiento

Los algoritmos de ordenamiento permiten reorganizar una colección de elementos de acuerdo con un criterio definido. Los más conocidos por su simplicidad son:

- Bubble Sort: compara elementos adyacentes y los intercambia si están en el orden incorrecto. Es útil para listas pequeñas, pero su complejidad es $O(n^2)$.
- Insertion Sort: construye la lista ordenada de forma incremental insertando elementos uno por uno. Es eficiente con listas pequeñas parcialmente ordenadas.
- Selection Sort: selecciona el menor elemento y lo posiciona correctamente, repitiendo el proceso con el resto de la lista.

2.2 Algoritmos de Búsqueda

Los algoritmos de búsqueda tienen como fin localizar un valor dentro de una colección de datos.

- Búsqueda lineal: recorre la lista elemento por elemento hasta encontrar el valor buscado. Su complejidad es $O(n)$ y funciona en listas no ordenadas.
- Búsqueda binaria: eficiente para listas ordenadas. Divide el espacio de búsqueda en mitades sucesivas hasta encontrar el elemento. Tiene complejidad $O(\log n)$. Su

versión iterativa es más eficiente en cuanto a memoria que la recursiva.

2.3 Comparación

Algoritmo	Complejidad temporal	Requiere lista ordenada
Búsqueda Lineal	$O(n)$	No
Búsqueda Binaria	$O(\log n)$	Sí
Bubble Sort	$O(n^2)$	No aplica

3. Caso Práctico: Gestión de pacientes hospitalarios

3.1 Descripción del problema

Se simula un sistema de gestión de pacientes donde cada paciente tiene nombre, DNI y nivel de prioridad (de 1 a 5, donde 5 es mayor urgencia). El sistema debe:

- Ordenar los pacientes por prioridad (de mayor a menor).
- Permitir buscar un paciente específico por DNI utilizando búsqueda binaria.

3.2 Código fuente

Aquí presentamos el código implementado en Python con comentarios explicativos:

```
pacientes = [  
    {"nombre": "Ana Gómez", "dni": 30123123, "prioridad": 3},  
    {"nombre": "Carlos Ruiz", "dni": 29500123, "prioridad": 5},  
    {"nombre": "Lucía Díaz", "dni": 33100222, "prioridad": 1},  
    {"nombre": "Marcos Ledesma", "dni": 31200123, "prioridad": 4},  
    {"nombre": "Fernanda Luna", "dni": 30765432, "prioridad": 2}  
]
```

Ordenamiento por prioridad (Bubble Sort adaptado)

```
def ordenar_por_prioridad(lista):  
    n = len(lista)  
  
    for i in range(n):  
  
        for j in range(0, n - i - 1):  
  
            if lista[j]["prioridad"] < lista[j + 1]["prioridad"]:  
  
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

Ordenamiento por DNI para aplicar búsqueda binaria

```
def ordenar_por_dni(lista):  
    lista.sort(key=lambda x: x["dni"])
```

Búsqueda binaria por DNI

```
def busqueda_binaria_dni(lista, dni_buscado):  
  
    inicio = 0  
  
    fin = len(lista) - 1  
  
    while inicio <= fin:  
  
        medio = (inicio + fin) // 2  
  
        if lista[medio]["dni"] == dni_buscado:  
  
            return lista[medio]  
  
        elif lista[medio]["dni"] < dni_buscado:  
  
            inicio = medio + 1
```

```
else:

    fin = medio - 1

return None
```

Aplicación

```
print("📌 Lista de pacientes ordenada por prioridad (mayor a menor):")

ordenar_por_prioridad(pacientes)

for paciente in pacientes:

    print(f'{paciente["nombre"]} - DNI: {paciente["dni"]} - Prioridad: {paciente["prioridad"]}')


print("\n🔍 Búsqueda binaria por DNI:")

ordenar_por_dni(pacientes)

dni_buscado = 12345678

resultado = busqueda_binaria_dni(pacientes, dni_buscado)

if resultado:

    print(f'✅ Paciente encontrado: {resultado["nombre"]} - DNI: {resultado["dni"]} - Prioridad: {resultado["prioridad"]}')

else:

    print("❌ Paciente no encontrado.")
```

3.3 Capturas

-lista inicial-

```
codigo.py x
codigo.py > ...
45 # dni_buscado = 31200123
46 # resultado = busqueda_binaria_dni(pacientes, dni_buscado)
47
48 # if resultado:
49 #     print(f"✅ Paciente encontrado: {resultado['nombre']} - DNI: {resultado['dni']} - Prioridad: {resultado['prioridad']}")
50 # else:
51 #     print(f"❌ Paciente no encontrado.")
52
53 print("📋 Lista original de pacientes:")
54 for p in pacientes:
55     print(p)
56
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
_varias\utn\programacion\tp integrador\codigo.py'
📋 Lista original de pacientes:
{'nombre': 'Ana Gómez', 'dni': 30123123, 'prioridad': 3}
{'nombre': 'Carlos Ruiz', 'dni': 29500123, 'prioridad': 5}
{'nombre': 'Lucía Díaz', 'dni': 33100222, 'prioridad': 1}
{'nombre': 'Marcos Ledesma', 'dni': 31200123, 'prioridad': 4}
{'nombre': 'Fernanda Luna', 'dni': 30765432, 'prioridad': 2}
PS D:\cosas_varias\utn\programacion\tp integrador>
```

Mostrando la lista original de pacientes, antes de realizar el ordenamiento de mayor a menor.

-Ordenado por prioridad de mayor a menor-

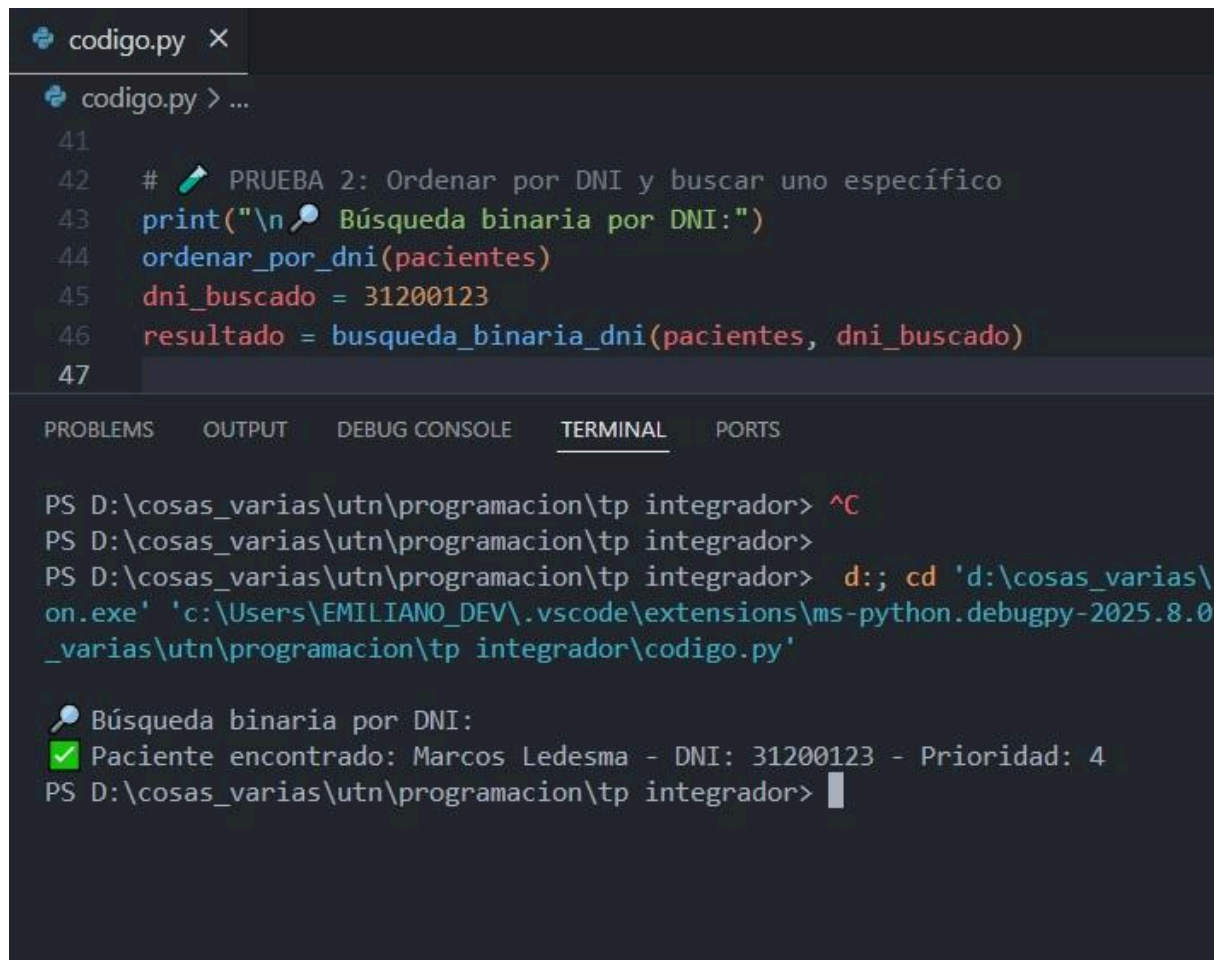
```
codigo.py x
codigo.py > ...
36 # # 🟢 PRUEBA 1: Ordenar por prioridad y mostrar resultado
37 print("🚀 Lista de pacientes ordenada por prioridad (mayor a menor):")
38 ordenar_por_prioridad(pacientes)
39 for paciente in pacientes:
40     print(f"{paciente['nombre']} - DNI: {paciente['dni']} - Prioridad: {paciente['prioridad']}")
41
42 # # 🟢 PRUEBA 2: Ordenar por DNI y buscar uno específico
43 # print("\n🔍 Búsqueda binaria por DNI:")

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
PS D:\cosas_varias\utn\programacion\tp integrador> ^C
PS D:\cosas_varias\utn\programacion\tp integrador>
PS D:\cosas_varias\utn\programacion\tp integrador> d:; cd 'd:\cosas_varias\utn\programacion\tp integrador\codigo.py'
🚀 Lista de pacientes ordenada por prioridad (mayor a menor):
Carlos Ruiz - DNI: 29500123 - Prioridad: 5
Marcos Ledesma - DNI: 31200123 - Prioridad: 4
Ana Gómez - DNI: 30123123 - Prioridad: 3
Fernanda Luna - DNI: 30765432 - Prioridad: 2
Lucía Díaz - DNI: 33100222 - Prioridad: 1
PS D:\cosas_varias\utn\programacion\tp integrador>
```

-Búsqueda binaria-



The image shows a VS Code editor window with a file named 'codigo.py'. The code in the editor is as follows:

```
41
42 # PRUEBA 2: Ordenar por DNI y buscar uno específico
43 print("\n Búsqueda binaria por DNI:")
44 ordenar_por_dni(pacientes)
45 dni_buscado = 31200123
46 resultado = busqueda_binaria_dni(pacientes, dni_buscado)
47
```

Below the editor, the 'TERMINAL' tab is active, showing the command prompt output:

```
PS D:\cosas_varias\utn\programacion\tp integrador> ^C
PS D:\cosas_varias\utn\programacion\tp integrador>
PS D:\cosas_varias\utn\programacion\tp integrador> d:; cd 'd:\cosas_varias\
on.exe' 'c:\Users\EMILIANO_DEV\.vscode\extensions\ms-python.debugpy-2025.8.0
_varias\utn\programacion\tp integrador\codigo.py'

 Búsqueda binaria por DNI:
✅ Paciente encontrado: Marcos Ledesma - DNI: 31200123 - Prioridad: 4
PS D:\cosas_varias\utn\programacion\tp integrador>
```

En esta imagen se muestra al paciente encontrado Marcos Ledesma, demostrando que el algoritmo de búsqueda binaria, es funcional.

-Paciente no encontrado-

```
codigo.py X
codigo.py > ...
40 # print(f"{paciente['nombre']} - DNI: {paciente['dni']}")
41
42 # PRUEBA 2: Ordenar por DNI y buscar uno específico
43 print("\n Búsqueda binaria por DNI:")
44 ordenar_por_dni(pacientes)
45 dni_buscado = 12345678
46 resultado = busqueda_binaria_dni(pacientes, dni_buscado)
47
48 if resultado:
49     print(f"✅ Paciente encontrado: {resultado['nombre']}")
50 else:
51     print("❌ Paciente no encontrado.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\cosas_varias\utn\programacion\tp integrador> ^C
PS D:\cosas_varias\utn\programacion\tp integrador>
PS D:\cosas_varias\utn\programacion\tp integrador> d:; cd 'd:\cosas_varias\utn\programacion\tp integrador'
PS D:\cosas_varias\utn\programacion\tp integrador> python .\codigo.py

Búsqueda binaria por DNI:
❌ Paciente no encontrado.
PS D:\cosas_varias\utn\programacion\tp integrador>
```

En esta imagen demostramos que retorna la función si el dni ingresado es invalido y no corresponde a ningún paciente registrado.

3.4 Justificación

Se eligió Bubble Sort por su sencillez didáctica, aunque en escenarios reales sería reemplazado por algoritmos más eficientes. La búsqueda binaria fue elegida por su alta eficiencia en grandes volúmenes de datos, siempre que la lista esté previamente ordenada.

Metodología Utilizada

El desarrollo del trabajo se estructuró en las siguientes etapas:

- Revisión teórica: Se consultaron libros, documentación oficial y recursos digitales para comprender a fondo los algoritmos.
- Diseño del caso práctico: Se planteó un escenario realista aplicable al ámbito de la salud, considerando criterios de urgencia.
- Codificación: Se programó el sistema en Python utilizando funciones personalizadas para el ordenamiento y búsqueda.
- Pruebas: Se verificó el funcionamiento correcto con diferentes DNIs y prioridades.
- Documentación: Se elaboró este informe y se registró el proyecto en un repositorio Git.

Herramientas utilizadas:

- Python 3.11
- Visual Studio Code
- GitHub para control de versiones

5. Resultados Obtenidos

- Se logró ordenar correctamente la lista de pacientes por prioridad.
- La búsqueda binaria identificó correctamente al paciente buscado por DNI.
- El programa respondió correctamente ante búsquedas exitosas y fallidas.
- Se comprobó la importancia de tener la lista ordenada para que la búsqueda binaria funcione.
- El código fue subido a GitHub y documentado.

Casos de prueba adicionales:

- DNI inexistente: retorna None.
- Lista vacía: búsqueda retorna None sin errores.
- Lista con elementos duplicados: el algoritmo retorna el primero que encuentra.

6. Conclusiones

Este trabajo nos permitió aplicar conceptos clave de programación en un caso concreto con impacto social real. Se comprendió en profundidad cómo los algoritmos de ordenamiento y búsqueda pueden mejorar procesos administrativos, como en un hospital.

La principal lección fue la importancia de elegir el algoritmo adecuado según el contexto, la estructura de los datos y el volumen de información. Además, se fortalecieron habilidades de análisis, resolución de problemas y trabajo colaborativo en equipo.

7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation.
<https://docs.python.org/3/>
- Khan Academy. (2024). Algoritmos de búsqueda y ordenamiento.
<https://es.khanacademy.org>
- Material otorgado por la cátedra.

8. Anexos

- Capturas de pantalla del código funcionando.
- Código fuente completo en archivo "codigo.py".
- Enlace al video explicativo: https://youtu.be/dzJI5kr0GP4?si=fWxpRFWnFWhb_qV0