Trabajo Integrador – Programación I

- **Título del trabajo**: Algoritmos de Búsqueda y Ordenamiento en Python: Aplicaciones y Comparaciones Simples
- **Alumno**: Cristian Darío Gómez – cristian.gomez@tupad.utn.edu.ar
- **Materia**: Programación I
- **Profesor/a**: Ariel Enferrel
- **Fecha de Entrega**: 09/06/2025

Índice

1. Introducción

El presente trabajo tiene como objetivo explorar el funcionamiento de los algoritmos de búsqueda y ordenamiento en el lenguaje de programación Python. Estos algoritmos representan herramientas fundamentales en el desarrollo de software, ya que permiten organizar y recuperar información de forma eficiente, dos aspectos clave en cualquier sistema informático.

La elección de este tema responde a su importancia formativa dentro del campo de la programación: entender cómo funcionan estos algoritmos permite a los programadores tomar mejores decisiones al momento de diseñar soluciones, optimizar el rendimiento de sus aplicaciones y comprender el comportamiento interno de muchas estructuras y funciones integradas.

El trabajo busca alcanzar los siguientes objetivos: presentar los conceptos teóricos principales, implementar ejemplos funcionales en Python, comparar su comportamiento y reflexionar sobre su utilidad en el desarrollo de software real. A través de este enfoque práctico, se espera consolidar los conocimientos adquiridos y desarrollar habilidades aplicables a proyectos futuros.

---

2. Marco Teórico

¿Qué es un algoritmo?

Un algoritmo es un conjunto finito y ordenado de instrucciones que, al ser ejecutadas paso a paso, permiten resolver un problema específico o realizar una tarea determinada. En programación, los algoritmos son la base del desarrollo de software y se implementan mediante lenguajes como Python, permitiendo automatizar procesos y manipular datos de manera eficiente.

Algoritmos de Búsqueda

Los algoritmos de búsqueda permiten encontrar un elemento específico dentro de una estructura de datos, como una lista.

- **Búsqueda Lineal**

Consiste en recorrer todos los elementos de una lista uno por uno hasta encontrar el valor buscado o llegar al final de la estructura.

```python
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

Ventajas: Simple de implementar.

Desventajas: Ineficiente en listas grandes.

Complejidad: O(n)

- **Búsqueda Binaria**

Requiere que la lista esté ordenada. Divide la lista a la mitad en cada iteración, comparando el elemento medio con el objetivo.

```python
def busqueda_binaria(lista, objetivo):
    inicio, fin = 0, len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1
```

Ventajas: Muy rápida en listas grandes ordenadas.

Desventajas: No sirve en listas desordenadas.

Complejidad: O(log n)

Algoritmos de Ordenamiento

Sirven para reorganizar una colección de elementos (como números o textos) en un orden determinado (ascendente o descendente).

- **Ordenamiento Burbuja (Bubble Sort)**

Compara pares de elementos y los intercambia si están en el orden incorrecto. Repite el proceso hasta que la lista esté ordenada.

```python
def burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

Sencillo, pero lento para listas grandes.

Complejidad: $O(n^2)$

- **Ordenamiento por Selección (Selection Sort)**

Selecciona el valor mínimo de la lista no ordenada y lo coloca al inicio.

```python
def seleccion(lista):
    for i in range(len(lista)):
        minimo = i
        for j in range(i + 1, len(lista)):
            if lista[j] < lista[minimo]:
                minimo = j
        lista[i], lista[minimo] = lista[minimo], lista[i]
```

- **Ordenamiento por Inserción (Insertion Sort)**

Inserta cada elemento en su lugar correcto en una sublista ordenada.

```python
def insercion(lista):
    for i in range(1, len(lista)):
        actual = lista[i]
        j = i - 1
        while j >= 0 and actual < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual
```

Comparación General (Tabla)

Algoritmo	Eficiencia Promedio	Lista Ordenada	Simplicidad	Ideal para...
Búsqueda Lineal	O(n)	No	Alta	Cualquier lista
Búsqueda Binaria	O(log n)	Sí	Media	Listas ordenadas
Ordenamiento Burbuja	$O(n^2)$	No	Alta	Enseñanza / demostración
Selección / Inserción	$O(n^2)$	No	Alta	Listas pequeñas

---

3. Caso Práctico

- **Descripción del problema**

  Se requiere gestionar una pequeña lista de estudiantes, permitiendo:

  - Buscar un estudiante por su número de documento (DNI).
  - Ordenar la lista por nota final de manera ascendente.
  - Buscar por DNI usando búsqueda binaria (tras ordenar por DNI).

- **Código Fuente**

codigo_fuente.py

```python
# Lista de estudiantes (cada uno es un diccionario)
estudiantes = [
    {"nombre": "Ana", "dni": 43123456, "nota_final": 8.5},
    {"nombre": "Luis", "dni": 40987654, "nota_final": 7.0},
    {"nombre": "Marta", "dni": 42111222, "nota_final": 9.2},
    {"nombre": "Pedro", "dni": 40123456, "nota_final": 6.8},
]

# Búsqueda lineal por DNI
def busqueda_lineal_por_dni(lista, dni):
    for estudiante in lista:
        if estudiante["dni"] == dni:
            return estudiante
    return None

# Ordenamiento burbuja por nota_final
def ordenar_por_nota(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j]["nota_final"] > lista[j + 1]["nota_final"]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]

# Búsqueda binaria por DNI (requiere lista ordenada por DNI)
def busqueda_binaria_por_dni(lista, dni):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        actual = lista[medio]["dni"]
        if actual == dni:
            return lista[medio]
```

Python

```
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}

Estudiante con DNI 40123456 (binaria):
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}

Criss@DESKTOP-NJBHTO6 MINGW64 ~/Documents/Facu/Programacion 1/UTN
-TUPaD-P1/Trabajo Integrador (main)
$ C:/Users/Criss/AppData/Local/Microsoft/WindowsApps/python3.13.e
xe "c:/Users/Criss/Documents/Facu/Programacion 1/UTN-TUPaD-P1/Tra
bajo Integrador/codigo_fuente.py"
Lista original:
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}
{'nombre': 'Luis', 'dni': 40987654, 'nota_final': 7.0}
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}

Estudiante con DNI 42111222:
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}

Lista ordenada por nota:
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}
{'nombre': 'Luis', 'dni': 40987654, 'nota_final': 7.0}
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}

Lista ordenada por DNI para búsqueda binaria:
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}
{'nombre': 'Luis', 'dni': 40987654, 'nota_final': 7.0}
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}

Estudiante con DNI 40123456 (binaria):
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}

Criss@DESKTOP-NJBHTO6 MINGW64 ~/Documents/Facu/Programacion 1/UTN
-TUPaD-P1/Trabajo Integrador (main)
$
```

Ln 58, Col 55 Spaces: 4 UTF-8 CRLF {} Python 3.13.3 64-bit (Microsoft Store)

```python
        return lista[medio]
    elif actual < dni:
        inicio = medio + 1
    else:
        fin = medio - 1
return None

# Ejecución y validación del funcionamiento
print("Lista original:")
for e in estudiantes:
    print(e)

print("\nEstudiante con DNI 42111222:")
print(busqueda_lineal_por_dni(estudiantes, 42111222))

print("\nLista ordenada por nota:")
ordenar_por_nota(estudiantes)
for e in estudiantes:
    print(e)

print("\nLista ordenada por DNI para búsqueda binaria:")
estudiantes.sort(key=lambda x: x["dni"])
for e in estudiantes:
    print(e)

print("\nEstudiante con DNI 40123456 (binaria):")
print(busqueda_binaria_por_dni(estudiantes, 40123456))
```

```
-TUPaD-P1/Trabajo Integrador (main)
$ C:/Users/Criss/AppData/Local/Microsoft/WindowsApps/python3.13.e
xe "c:/Users/Criss/Documents/Facu/Programacion 1/UTN-TUPaD-P1/Tra
bajo Integrador/codigo_fuente.py"
Lista original:
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}
{'nombre': 'Luis', 'dni': 40987654, 'nota_final': 7.0}
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}

Estudiante con DNI 42111222:
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}

Lista ordenada por nota:
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}
{'nombre': 'Luis', 'dni': 40987654, 'nota_final': 7.0}
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}

Lista ordenada por DNI para búsqueda binaria:
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}
{'nombre': 'Luis', 'dni': 40987654, 'nota_final': 7.0}
{'nombre': 'Marta', 'dni': 42111222, 'nota_final': 9.2}
{'nombre': 'Ana', 'dni': 43123456, 'nota_final': 8.5}

Estudiante con DNI 40123456 (binaria):
{'nombre': 'Pedro', 'dni': 40123456, 'nota_final': 6.8}

Criss@DESKTOP-NJBHTO6 MINGW64 ~/Documents/Facu/Programacion 1/UTN
-TUPaD-P1/Trabajo Integrador (main)
```

- **Decisiones de diseño**
  - Se usó **burbuja** por su simplicidad para explicar en video.
  - Se implementó **búsqueda binaria** solo tras ordenar la lista por DNI.
  - Se manejó la estructura de datos con listas de diccionarios para facilitar la lectura.

---

4. Metodología Utilizada

El desarrollo se llevó a cabo en varias etapas, combinando investigación teórica y aplicación práctica:

- **Investigación previa:** Se consultaron libros, documentación oficial de Python y materiales académicos sobre algoritmos clásicos de búsqueda y ordenamiento.
- **Diseño del caso práctico:** Se eligió simular un sistema de gestión de estudiantes, representando cada alumno como un diccionario dentro de una lista.
- **Desarrollo y prueba del código:** Se implementaron algoritmos de búsqueda lineal, búsqueda binaria (con ordenamiento previo), y ordenamiento burbuja. Se utilizó Python 3.11 en el entorno de desarrollo VS Code.
- **Validación:** Se realizaron pruebas de funcionamiento para asegurar la correcta búsqueda y ordenamiento de datos.
- **Documentación y presentación:** Se comentaron todos los fragmentos de código, se prepararon capturas de pantalla y se organizó el contenido según la plantilla oficial.

---

5. Resultados Obtenidos

El caso práctico permitió aplicar exitosamente los algoritmos teóricos estudiados. Los resultados fueron los siguientes:

- Se logró buscar correctamente estudiantes por DNI utilizando tanto búsqueda lineal como binaria (previa ordenación).
- Se ordenó la lista de estudiantes por nota final usando el algoritmo burbuja, y luego por DNI con el método sort() de Python para habilitar la búsqueda binaria.
- El código se ejecutó sin errores, mostrando los resultados esperados en consola.

- Se identificaron diferencias de eficiencia y condiciones de uso entre los algoritmos (por ejemplo, la búsqueda binaria solo fue posible tras ordenar la lista).
- Se subió el código funcional y documentado a un repositorio GitHub (enlace se incluye en la bibliografía o anexos).

---

6. Conclusiones

La realización del presente trabajo integrador permitió reforzar los conocimientos sobre algoritmos fundamentales de programación, en especial los relacionados con búsqueda y ordenamiento.

Se aprendió que la **elección del algoritmo adecuado depende del tipo de datos, la cantidad de elementos y el contexto del problema**. También se evidenció que, si bien algunos algoritmos como burbuja o búsqueda lineal son más simples, pueden resultar poco eficientes ante grandes volúmenes de datos.

El enfoque práctico ayudó a consolidar la teoría, facilitando la comprensión a través de ejemplos concretos. Además, trabajar en forma autónoma permitió ejercitar la planificación, la documentación del código y la capacidad de explicar el proceso de desarrollo con claridad.

Una posible mejora futura sería comparar empíricamente el rendimiento (tiempo de ejecución) entre distintos algoritmos, utilizando estructuras más grandes o herramientas como el módulo time.

---

7. Bibliografía

- Universidad Tecnológica Nacional – Facultad Regional Córdoba. (2022). *Apuntes de Programación I – Estructuras de Datos*. Recuperado de: https://frc.utn.edu.ar/
- Universidad Nacional del Litoral – Facultad de Ingeniería y Ciencias Hídricas. (2020). *Algoritmos y Estructuras de Datos*. Recuperado de: https://fich.unl.edu.ar/estructura-de-datos

- Universidad Nacional de La Plata – Facultad de Informática. (2019). *Fundamentos de Programación*. Disponible en: http://www.info.unlp.edu.ar/
- Fundación Python. (2024). *Documentación oficial de Python 3*. Recuperado de: https://docs.python.org/es/3/
- UTN – Facultad Regional Buenos Aires. (s.f.). *Curso de Python – Material de Apoyo para Programación I*. Recuperado de: https://www.frba.utn.edu.ar/

---

8. Anexos

- Link a repositorio de GitHub: https://github.com/kakamoto74/algoritmos-python-integrador.git

- Link a video de YouTube: https://youtu.be/k-7blGZxVc0