

Trabajo Integrador – Programación I

Tema: Algoritmos de Búsqueda y Ordenamiento

Datos Generales

📌 Título del trabajo: Algoritmos de Búsqueda y Ordenamiento en Python

👤 Alumnos:

- Stefan, Dios Mayarin - stefan.dios@tupad.utn.edu.ar

- Mathias, Flor - mathias.flor@tupad.utn.edu.ar

📖 Materia: Programación I

👨‍🏫 Profesor: AUS Bruselario, Sebastián

👩‍🎓 Tutora: Candia, Verónica

📅 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 integrador aborda el tema de los **algoritmos de búsqueda y ordenamiento**, un componente fundamental dentro del desarrollo de software y estructuras de datos.

Se eligió esta temática por su aplicabilidad directa a numerosos contextos reales, como por ejemplo, el manejo de listas de usuarios, el filtrado de resultados, o la clasificación de datos dentro de una aplicación. En particular, se buscó que el trabajo práctico sea accesible para quienes están dando sus primeros pasos en la programación, sin dejar de ser representativo del funcionamiento de estas técnicas.

Estos algoritmos tienen una **gran importancia en la programación** debido a que optimizan el rendimiento de los sistemas al reducir el tiempo de procesamiento y búsqueda. Su estudio también permite comprender cómo se diseñan programas eficientes y cómo se elige la mejor estrategia según el tipo y tamaño de los datos.

El objetivo principal del trabajo es aplicar los conceptos de búsqueda (lineal y binaria) y ordenamiento (burbuja) en un caso práctico simple y didáctico. Para ello, se desarrolla un sistema de gestión de estudiantes que permita cargar alumnos con su respectiva nota, ordenarlos por nombre o calificación, y buscar estudiantes por nombre.

De esta forma, se propone reforzar los conocimientos adquiridos a lo largo de la cursada, integrando teoría y práctica en un mismo proyecto.

2. Marco Teórico

Los algoritmos de búsqueda y ordenamiento son fundamentales en la programación, ya que permiten localizar y organizar información de forma eficiente, lo que influye directamente en el rendimiento de las aplicaciones. Su estudio permite comprender cómo optimizar el acceso a los datos, mejorar el tiempo de respuesta de los programas y utilizar mejor los recursos del sistema.

Algoritmos de Búsqueda

Definición

Un algoritmo de búsqueda permite encontrar un elemento dentro de una estructura de datos, como una lista. Dependiendo de su tipo, puede tener un rendimiento muy diferente.

Clasificación y ejemplos:

- **Búsqueda Lineal:**
 - Recorre cada elemento hasta encontrar el deseado.
 - Simple de implementar.
 - No necesita una lista ordenada.

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

Búsqueda Binaria:

- Más eficiente.
- Solo funciona con listas ordenadas.
- Divide la lista en mitades sucesivas.

```
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
```

Algoritmos de Ordenamiento

Definición

Permiten reordenar los datos en un orden determinado (ascendente o descendente), lo que facilita la búsqueda y la presentación de la información.

Clasificación y ejemplos:

- **Bubble Sort (Burbuja):**

- Compara elementos adyacentes e intercambia si están en orden incorrecto.
- Fácil de entender, pero lento en grandes volúmenes.

```
def bubble_sort(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]
```

🔗 Quick Sort:

- Utiliza recursividad y un elemento pivote.
- Muy eficiente en promedio.

```
def quick_sort(lista):  
    if len(lista) ≤ 1:  
        return lista  
    pivote = lista[0]  
    menores = [x for x in lista[1:] if x ≤ pivote]  
    mayores = [x for x in lista[1:] if x > pivote]  
    return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

🔗 Merge Sort:

- Divide y conquista.
- Divide la lista en mitades, las ordena y luego las combina.

```

def merge_sort(lista):
    if len(lista) ≤ 1:
        return lista
    medio = len(lista) // 2
    izquierda = merge_sort(lista[:medio])
    derecha = merge_sort(lista[medio:])
    return merge(izquierda, derecha)

def merge(izq, der):
    resultado = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1
    resultado += izq[i:]
    resultado += der[j:]
    return resultado

```

Implementación en Python

A continuación se presenta un ejemplo práctico de implementación de los algoritmos de búsqueda y ordenamiento usando una lista de estudiantes. Esta imagen representa el código real usado en el caso práctico **(3. Caso Práctico: Gestión de Estudiantes)**

3. Caso Práctico: Gestión de Estudiantes

Este caso práctico consiste en simular una gestión básica de estudiantes mediante una lista de diccionarios en Python, donde se almacenan sus nombres y notas. Se aplican dos algoritmos clásicos: búsqueda lineal y ordenamiento burbuja.

Descripción del Problema

Se necesita ordenar una lista de estudiantes por sus notas (de menor a mayor) y permitir al usuario buscar un estudiante por su nombre para obtener su nota.

Código Fuente Comentado

```

# Lista de estudiantes, cada uno con su nombre y nota
estudiantes = [
    {"nombre": "Ana", "nota": 7},
    {"nombre": "Luis", "nota": 5},
    {"nombre": "Marta", "nota": 9},
    {"nombre": "Carlos", "nota": 6},
    {"nombre": "Julia", "nota": 8}
]

# FUNCION DE ORDENAMIENTO: Burbuja (Bubble Sort)
# Ordena la lista por nota de menor a mayor
def ordenar_burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            # Si la nota actual es mayor que la siguiente, se
            intercambian
            if lista[j]["nota"] > lista[j + 1]["nota"]:
                lista[j], lista[j + 1] = lista[j + 1],
                lista[j]
    return lista

# FUNCION DE BÚSQUEDA: Búsqueda lineal
# Busca un estudiante por nombre
def busqueda_lineal(lista, nombre_buscado):
    for estudiante in lista:
        if estudiante["nombre"].lower() ==
        nombre_buscado.lower():
            return estudiante
    return None

# Mostrar la lista original
print("Lista original de estudiantes:")
for est in estudiantes:
    print(f"{est['nombre']} - Nota: {est['nota']}")

# Ordenar la lista y mostrarla
print("\n Lista ordenada por nota (de menor a mayor):")
ordenada = ordenar_burbuja(estudiantes.copy())
for est in ordenada:
    print(f"{est['nombre']} - Nota: {est['nota']}")

# Pedir al usuario un nombre para buscar
nombre = input("\n Ingresá el nombre del estudiante a buscar:
")

```

```

resultado = busqueda_lineal(estudiantes, nombre)

# Mostrar el resultado de la búsqueda
if resultado:
    print(f"\n Estudiante encontrado: {resultado['nombre']} -
Nota: {resultado['nota']}")
else:
    print("\n Estudiante no encontrado.")

```

Decisiones de Diseño

Se eligió el algoritmo de **ordenamiento burbuja** por su simplicidad y claridad, ideal para fines didácticos y listas pequeñas. La **búsqueda lineal** también fue seleccionada por no requerir que la lista esté ordenada, facilitando su implementación.

Validación

El programa fue ejecutado y validado con diferentes entradas. El ordenamiento mostró correctamente los estudiantes de menor a mayor nota, y la búsqueda identificó correctamente estudiantes existentes e inexistentes. Esto demuestra que los algoritmos funcionan según lo esperado.

Comparativa General

Algoritmo	Velocidad (Peor Caso)	Requiere lista ordenada	Uso de Memoria	Aplicación ideal
Búsqueda Lineal	$O(n)$	No	Baja	Listas pequeñas y simples
Búsqueda Binaria	$O(\log n)$	Sí	Baja	Listas grandes y ordenadas
Bubble Sort	$O(n^2)$	No	Baja	Fines didácticos
Quick Sort	$O(n \log n)$ promedio	No	Media	Listas grandes (muy eficiente)
Merge Sort	$O(n \log n)$	No	Alta	Listas grandes, más estable

4. Metodología Utilizada

Para llevar adelante este trabajo integrador se siguieron diferentes etapas de trabajo, investigación y desarrollo práctico, organizadas de la siguiente manera:

Investigación previa

Se consultaron múltiples fuentes de contenido, incluyendo:

- Clases teóricas y prácticas de la materia Programación I.
- Videos provistos por la cátedra (Introducción a Búsqueda y Ordenamiento, Ejemplos de implementación en Python).
- Archivos modelo y guías provistas por el aula virtual (plantillas, rúbricas y ejemplos de trabajos integradores).
- Documentación oficial de Python y recursos educativos como W3Schools y GeeksforGeeks.

Diseño y prueba del código

1. Se seleccionó un caso práctico simple y concreto: Gestión de estudiantes.
2. Se diseñó la estructura de datos utilizando listas de diccionarios en Python.
3. Se implementaron dos algoritmos fundamentales:
 - Ordenamiento burbuja (para organizar a los estudiantes por nota).
 - Búsqueda lineal (para localizar un estudiante por nombre).
4. Se probó el código con diferentes entradas para validar su correcto funcionamiento.

Herramientas utilizadas

- Lenguaje: Python
- IDE: Visual Studio Code (VSCode)

- **Extensiones:** Python Extension for VSCode, Code Runner.
- **Control de versiones:** GitHub, para alojar el código fuente y el archivo README.
- **Otros:** Word para la redacción del informe.

Trabajo colaborativo

El trabajo fue desarrollado en conjunto por los dos integrantes del grupo. Se acordó la siguiente distribución de tareas:

- **Mathias Flor:**
 - Redacción del informe y marco teórico.
 - Implementación del algoritmo de búsqueda lineal.
 - Presentación en video y edición final del documento.
- **Stefan, Dios Mayarin:**
 - Implementación del algoritmo de ordenamiento burbuja.
 - Validación del código y pruebas funcionales.
 - Capturas de pantalla y organización de carpeta digital y repositorio.

5. Resultados Obtenidos

Durante el desarrollo y ejecución del caso práctico, se lograron los siguientes resultados concretos:

Funcionamiento correcto

- El algoritmo de **ordenamiento burbuja** reordenó correctamente la lista de estudiantes según sus notas, de menor a mayor.
- La **búsqueda lineal** identificó con precisión si un estudiante ingresado por el usuario se encontraba en la lista, mostrando su nota correspondiente.
- El código respondió adecuadamente en todos los casos de prueba.

Casos de prueba

1. **Estudiante existente:** búsqueda de “Marta” → Se encontró correctamente con nota 9.
2. **Estudiante inexistente:** búsqueda de “Roberto” → Resultado: “Estudiante no encontrado.”
3. **Lista sin ordenar:** Se validó que la búsqueda funciona sin importar el orden inicial.
4. **Nombres con mayúsculas/minúsculas:** Se probó que no es sensible al uso de mayúsculas.

Errores corregidos

- Se corrigieron errores iniciales en la lógica del ordenamiento (índices mal posicionados).
- Se ajustó el algoritmo de búsqueda para que compare los nombres ignorando mayúsculas/minúsculas.


Evaluación de rendimiento

Dado que el volumen de datos es pequeño y el enfoque es educativo, no se aplicó medición precisa de rendimiento. Sin embargo:

- **Ordenamiento burbuja** tiene una complejidad de $O(n^2)$, adecuado solo para listas cortas.
- **Búsqueda lineal** también tiene $O(n)$, aceptable para la cantidad de estudiantes utilizada.

Enlace a Repositorio

El código fuente del caso práctico se encuentra disponible en el siguiente repositorio de GitHub:

 <https://github.com/stefanmdev/busqueda-ordenamiento-python>

6. Conclusiones

Durante la realización del trabajo integrador logramos reforzar los conceptos fundamentales de programación relacionados con algoritmos de búsqueda y ordenamiento. Pudimos aplicar de manera concreta conocimientos sobre estructuras de datos (listas), condicionales, bucles, funciones y validaciones.

Aprendimos cómo cada tipo de algoritmo tiene ventajas y limitaciones según el contexto, y cómo estas decisiones influyen en el rendimiento del programa. Además, comprendimos la utilidad de estos algoritmos en múltiples situaciones prácticas dentro de sistemas informáticos reales.

El trabajo colaborativo entre ambos integrantes permitió distribuir tareas de manera equitativa, lo que facilitó el desarrollo y revisión del código, así como la redacción del informe.

Como posibles mejoras futuras, podríamos integrar más tipos de algoritmos (como búsqueda binaria o quicksort), trabajar con volúmenes más grandes de datos, implementar interfaces gráficas y medir rendimiento con herramientas específicas.

Durante el proceso surgieron algunas dificultades, como errores en el ordenamiento por índices y validación de entrada del usuario. Estas fueron resueltas mediante pruebas, depuración paso a paso y consulta con el material de clase y fuentes externas.

En resumen, el trabajo fue una experiencia valiosa para afianzar nuestra lógica de programación y capacidad de resolución de problemas.



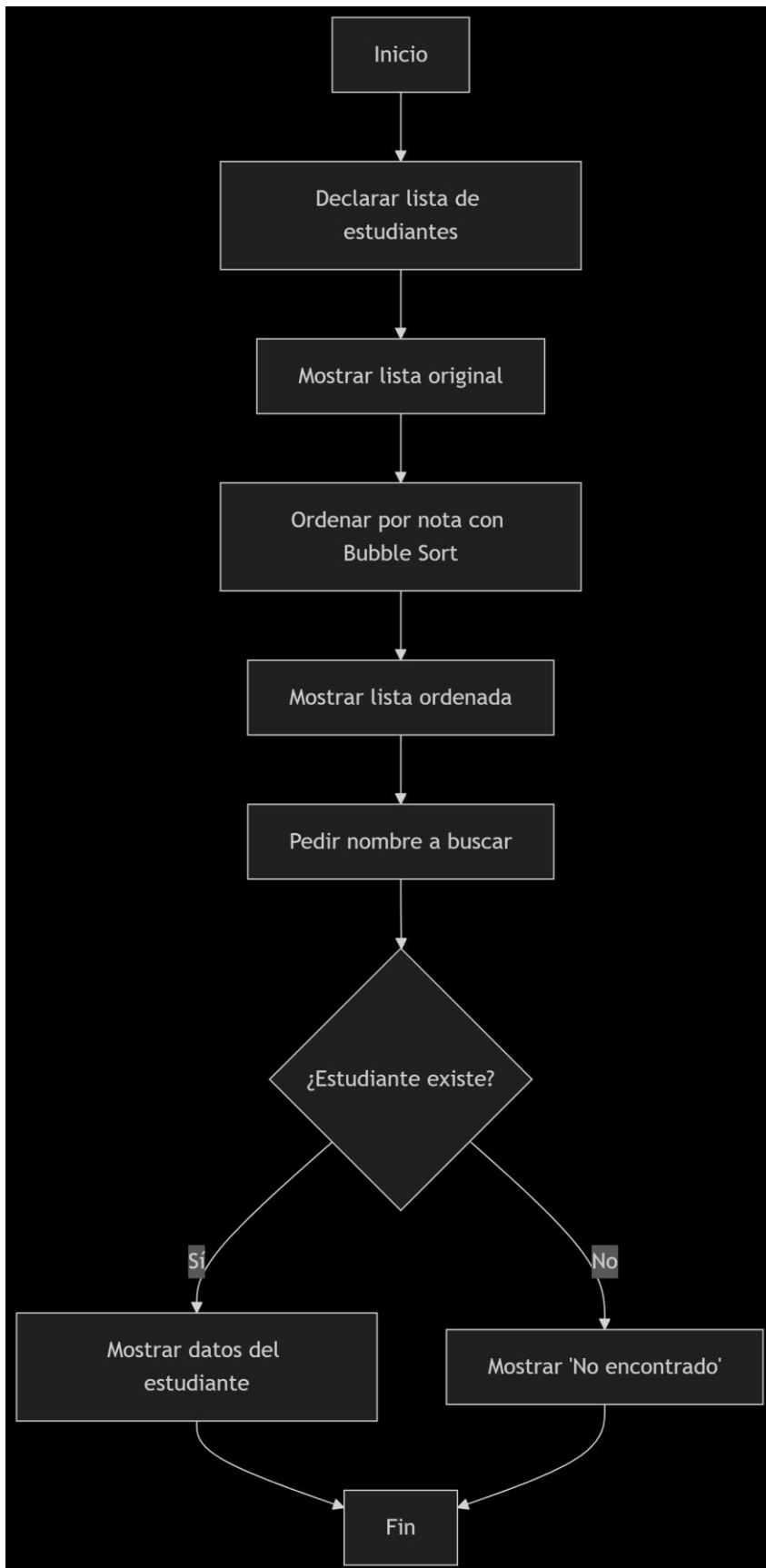
7. Bibliografía

- Python Software Foundation. (2024). *Python 3 Documentation*. Recuperado de <https://docs.python.org/3/>
- <https://www.youtube.com/watch?v=haF4P8kF4Ik>
- https://www.youtube.com/watch?v=u1QuRbx-_x4
- W3Schools. (2023). *Python Lists and Functions*. <https://www.w3schools.com/python/>
- <https://medium.com/@mise/algoritmos-de-b%C3%BAsqueda-y-ordenamiento-7116bcea03d0>



8. Anexos

A continuación, se incluye material complementario que respalda el desarrollo del trabajo integrador y aporta valor a su comprensión:



❑ 📷 Capturas del programa funcionando

Se adjuntan imágenes donde se observa el funcionamiento del código en la terminal:

- Lista original de estudiantes.
 - Lista ordenada por nota.
 - Resultado de búsqueda exitosa y no exitosa.
- (Ver imágenes incluidas en el documento)*

```
DELL@DELL-XPS13 MINGW64 ~/Desktop/Trabajo Integrador - PI
$ C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/DELL/Desktop/Trabajo Integrador - PI/BusquedaOrdenamiento.py"
Lista original de estudiantes:
Ana - Nota: 7
Luis - Nota: 5
Marta - Nota: 9
Carlos - Nota: 6
Julia - Nota: 8

Lista ordenada por nota (de menor a mayor):
Luis - Nota: 5
Carlos - Nota: 6
Carlos - Nota: 6
Ana - Nota: 7
Julia - Nota: 8
Marta - Nota: 9

Ingresá el nombre del estudiante a buscar: Julia

Estudiante encontrado: Julia - Nota: 8
```

```
DELL@DELL-XPS13 MINGW64 ~/Desktop/Trabajo Integrador - PI
$ C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/DELL/Desktop/Trabajo Integrador - PI/BusquedaOrdenamiento.py"
Lista original de estudiantes:
Ana - Nota: 7
Luis - Nota: 5
Marta - Nota: 9
Carlos - Nota: 6
Julia - Nota: 8

Lista ordenada por nota (de menor a mayor):
Luis - Nota: 5
Carlos - Nota: 6
Ana - Nota: 7
Julia - Nota: 8
Marta - Nota: 9

Ingresá el nombre del estudiante a buscar: Roberto

Estudiante no encontrado.
```

