Algoritmos de Búsqueda y Ordenamiento en Python

• **Título del trabajo**: “Algoritmos de Búsqueda y Ordenamiento en Python”

• **Alumnos**:

- Eliud Campos
- Rebeca A. Coletti

• **Materia**: Programación I

• **Comisión: 11**

1. Introducción

El tema abordado en este trabajo son los algoritmos, herramientas fundamentales en el desarrollo de programas, que permiten resolver problemas o realizar tareas de manera ordenada, lógica y eficiente.

Por esta razón, se eligió trabajar con algoritmos de búsqueda y ordenamiento, ya que tienen un papel central en la formación de programadores.

En el contexto de la Tecnicatura Universitaria en Programación, comprender el funcionamiento de estos algoritmos no solo mejora la lógica de programación y el uso adecuado de estructuras de datos, sino que también contribuye al desarrollo del pensamiento algorítmico, la capacidad de análisis y el criterio para la toma de decisiones técnicas.

Este trabajo tiene como objetivo principal que el estudiante:

- Comprenda la lógica y la eficiencia de distintos algoritmos de ordenamiento (Bubble Sort, Merge Sort y QuickSort) y de búsqueda (lineal y binaria).
- Adquiera experiencia en su implementación práctica en Python.
- Evalúe su comportamiento a través de ejemplos concretos y casos de prueba.
- Analice sus ventajas, desventajas y aplicaciones posibles según el tipo y tamaño de datos.

El propósito general es afianzar conocimientos teóricos y prácticos que resultarán fundamentales en materias futuras y en entornos reales de programación.

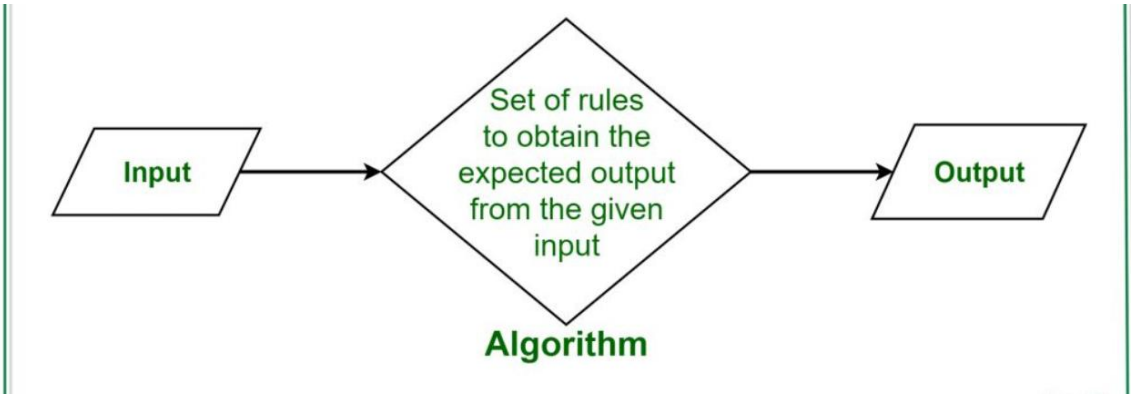
2. Marco Teórico

En programación, se usan algoritmos para decirle a la computadora qué hacer.

Un **algoritmo**[1] es una secuencia finita de instrucciones bien definidas que se pueden utilizar para resolver un problema computacional. Proporciona un procedimiento paso a paso que convierte una entrada en una salida deseada.

Los algoritmos suelen seguir una estructura lógica:

- **Entrada:** El algoritmo recibe datos de entrada.
- **Procesamiento:** El algoritmo realiza una serie de operaciones sobre los datos de entrada.
- **Salida:** El algoritmo produce la salida deseada.



En programación, es muy común tener que **ordenar datos** (por ejemplo, de menor a mayor) o **buscar un valor específico** dentro de una lista. Para hacer estas tareas de manera rápida y eficiente, usamos **algoritmos de búsqueda y ordenamiento.**

Estos algoritmos son herramientas fundamentales para la programación, ya que optimizan el tiempo de procesamiento, mejoran el rendimiento de los programas y que permiten automatizar procedimientos.

Ahora bien **¿Qué es ordenar y buscar?** para comprender mejor estos conceptos lo mejor es utilizar analogías de la vida cotidiana.

---

[1] Tutorial de algoritmos | GeeksforGeeks

Ordenar es una acción presente tanto en la vida cotidiana como en el mundo de la programación. En lo cotidiano, ordenar significa organizar elementos según un criterio específico para facilitar su uso o acceso. Por ejemplo, al acomodar ropa por tipo o color, o al clasificar libros por tamaño o temática, se logra una mejor organización que permite encontrar lo buscado con mayor rapidez.

Del mismo modo, en programación, el ordenamiento organiza los datos de acuerdo con un criterio, como de menor a mayor o alfabéticamente.

Entonces ¿Qué papel juegan los algoritmos en este proceso? Los algoritmos de ordenamiento son herramientas fundamentales para estructurar los datos de manera óptima. Permiten realizar búsquedas más eficientes, simplificar el análisis de datos y acelerar diversas operaciones.

Por su parte buscar es otra acción que acción que realizamos constantemente en la vida cotidiana, como cuando intentamos encontrar una palabra en un libro, una dirección en un mapa o una foto específica en el celular. Esta acción consiste en localizar un elemento determinado dentro de un conjunto más amplio.

En el ámbito de la programación, la búsqueda cumple exactamente ese mismo rol, pero aplicada a estructuras de datos como listas, arreglos, tablas o bases de datos. Es una operación fundamental que permite encontrar un valor específico dentro de un conjunto de datos, siendo esencial en múltiples áreas, como bases de datos, sistemas de archivos, motores de búsqueda e incluso en algoritmos de inteligencia artificial.

Cada tipo de algoritmo tiene ventajas y desventajas, dependiendo del contexto y de cómo estén organizados los datos.

Por ello el desarrollo de criterio por parte del programador es esencial desde sus primeros pasos.

Los Ordenamientos por tratar tienen diferentes características que se verán a continuación:

Bubble Sort	Merge Sort	QuickSort
Compara elementos **adyacentes** y los intercambia si están en orden incorrecto.	**Divide** el arreglo en mitades, ordena cada mitad y luego las **fusiona**.	Elige un **pivote**, divide el arreglo en menores y mayores al pivote, y ordena cada parte **recursivamente**.
También se conoce como **ordenamiento burbuja**.	También se llama **ordenamiento por mezcla**.	También se conoce como **ordenamiento rápido**.
Tiene complejidad temporal **O(n²)** en el peor caso.	Tiene complejidad temporal **O(n log n)** en todos los casos.	Tiene complejidad temporal **O(n log n)** en el promedio, pero **O(n²)** en el peor caso.
Es **muy ineficiente** para listas largas.	Es **eficiente y estable**, ideal para grandes cantidades de datos.	Es **muy eficiente en la práctica**, especialmente con buenos pivotes.
No requiere estructura auxiliar adicional.	Requiere espacio extra para fusionar subarreglos.	No requiere espacio adicional si se implementa en el lugar (**in-place**).
Es **fácil de implementar**, ideal para aprendizaje inicial.	Es **más complejo de implementar**, pero muy robusto.	Tiene **complejidad media** en implementación, pero muy usado.
No es adecuado para grandes volúmenes de datos.	Adecuado para **datos grandes y estables**.	Excelente para **datos no muy ordenados** o aleatorios.

Por su parte las búsquedas[2] que se analizan tienen las siguientes características:

[2] Búsqueda lineal vs búsqueda binaria | GeeksforGeeks

Búsqueda lineal	Búsqueda binaria
En la búsqueda lineal, los datos de entrada no tienen por qué estar ordenados.	En la búsqueda binaria, los datos de entrada deben estar ordenados.
También se denomina búsqueda secuencial.	También se denomina búsqueda de medio intervalo.
La complejidad temporal de la búsqueda lineal **O(n)**.	La complejidad temporal de la búsqueda binaria **O(log n)**.
Se puede utilizar una matriz multidimensional.	Solo se utiliza una matriz unidimensional.
La búsqueda lineal realiza comparaciones de igualdad	La búsqueda binaria realiza comparaciones de ordenación
Es menos complejo.	Es más complejo.
Es un proceso muy lento.	Es un proceso muy rápido.

3. Caso Practico:

El objetivo de este trabajo es demostrar el desarrollo de una aplicación por consola en Python que permita gestionar una lista de películas ingresadas por el usuario. El sistema incluye algoritmos de ordenamiento y búsqueda, y permite la comparación entre distintos métodos.

Como parte del caso práctico, se agregó un ejemplo especial con **una lista larga de 100 películas precargadas automáticamente**, para analizar el comportamiento de los algoritmos de búsqueda y ordenamiento en un contexto más exigente.

Se presenta el código comentado para su análisis:

```python
import time  # Importamos el modulo 'time' para poder medir cuanto tarda cada proceso (ordenar y buscar)

def ejecutar_ejemplo_lista_larga():
    print("\n=== EJEMPLO CON LISTA LARGA ===")  # Indicamos el inicio de este ejemplo especial
    print(" Generando automáticamente 100 películas...")  # Le informamos al usuario lo que vamos a hacer

    # Generamos una lista con 100 peliculas. Cada pelicula es un diccionario con titulo, año y director.
    # El titulo va de 'Pelicula 001' a 'Pelicula 100'
    # El año va de 1980 y 2019
    # El director va de la A-Z
    peliculas = [
        {"titulo": f"Pelicula {i+1:03}", "anio": 1980 + (i % 40), "director": f"Director {chr(65 + i % 26)}"}
        for i in range(100)
    ]

    # Definimos con que clave/dato vamos a trabajar: en este caso, el 'titulo' de las peliculas
    clave = "titulo"

    print(" Por el tamaño de la lista, se ordenará automáticamente con QuickSort.")  # Le explicamos al usuario por qué QuickSort

    # MEDICIÓN DEL TIEMPO DE ORDENAMIENTO
    start_sort = time.time()  # Empezamos a contar el tiempo antes de ordenar

    # Ordenamos la lista usando QuickSort (definido en peliculas_utils.py), guardamos:
    # la lista ordenada
    # los pasos
    # la cantidad de comparaciones
    # la cantidad de intercambios realizados
    ordenadas, pasos, comps, intercambios = quicksort_peliculas(peliculas, clave)

    end_sort = time.time()  # Terminamos de contar el tiempo
    tiempo_sort = end_sort - start_sort  # Calculamos cuanto tiempo tarda el proceso de ordenamiento

    # Mostramos las primeras 10 peliculas ordenadas, como una vista previa (elegimos que sean 10 ya que la lista entera es muy larga)
    print(f"\n Primeras 10 películas ordenadas por {clave}:")
    for i, p in enumerate(ordenadas[:10]):
        print(f"{i+1}. {p['titulo']} ({p['anio']}) - {p['director']}")

    # vamos a mostrar cuantas comparaciones e intercambios se hicieron al ordenar
    print(f"\nComparaciones: {comps} | Intercambios: {intercambios}")
```

```python
print("\n Ahora se buscará una película con búsqueda binaria (Pelicula 050)...")  # Informamos qué va a hacer el programa ahora

# MEDICIÓN DEL TIEMPO DE BÚSQUEDA
start_search = time.time()  # Empezamos a contar el tiempo antes de buscar

# Realizamos la busqueda binaria sobre la lista ordenada.
# Buscamos la pelicula 'Pelicula 050'
# Nos devuelve el índice donde está, los pasos realizados y la cantidad de comparaciones
idx, pasos_busq, comps_busq = binary_search_peliculas(ordenadas, "Pelicula 050", clave)

end_search = time.time()  # Finalizamos la medición de tiempo
tiempo_search = end_search - start_search  # Calculamos cuanto tarda la busqueda

# Si se encontró la película, el programa la mostrara completa; si no, indicara que no fue hallada
if idx != -1:
    peli = ordenadas[idx]
    print(f"\n Película encontrada: {peli['titulo']} ({peli['anio']}) - {peli['director']}")
else:
    print("\n Película no encontrada.")

# tambien vamos a mostramos cuantas comparaciones fueron necesarias en la busqueda
print(f"Comparaciones en búsqueda: {comps_busq}")

# Mostramos el detalle de los pasos realizados durante la búsqueda
print("Pasos del algoritmo:")
for paso in pasos_busq:
    print(paso)

# POR ULTIMO PASAMOS A UN RESUMEN FINAL: esta parte nos parecio importante para explicar por qué elegimos QuickSort y búsqueda binaria en este caso
print("\n RESUMEN:")
print("Se ordenó con el método QuickSort porque la lista es muy grande (100 elementos),")
print("lo cual lo hace más eficiente que otros algoritmos como Bubble Sort.")
print("Se realizó la búsqueda con el método binario porque es más rápido que la búsqueda lineal en listas ordenadas.")

# Finalmente, mostramos cuánto tiempo tardó cada uno de los dos procesos (ordenar y buscar)
print(f"\n Tiempo de ordenamiento: {tiempo_sort:.6f} segundos")
print(f" Tiempo de búsqueda: {tiempo_search:.6f} segundos")
```

Cuando el usuario elige la opción 5, el código da como resultado:

```
=== Menú de Películas ===
1. Ingresar lista de películas
2. Ordenar películas
3. Buscar película
4. Guardar último resultado de ordenación
5. Ejecutar ejemplo con lista larga (100 películas precargadas)
0. Salir
Elige una opción: 5

=== EJEMPLO CON LISTA LARGA ===
 Generando automáticamente 100 películas...
 Por el tamaño de la lista, se ordenará automáticamente con QuickSort.

 Primeras 10 películas ordenadas por titulo:
1. Pelicula 001 (1980) - Director A
2. Pelicula 002 (1981) - Director B
3. Pelicula 003 (1982) - Director C
4. Pelicula 004 (1983) - Director D
5. Pelicula 005 (1984) - Director E
6. Pelicula 006 (1985) - Director F
7. Pelicula 007 (1986) - Director G
8. Pelicula 008 (1987) - Director H
9. Pelicula 009 (1988) - Director I
10. Pelicula 010 (1989) - Director J

Comparaciones: 4950 | Intercambios: 4950

 Ahora se buscará una película con búsqueda binaria (Pelicula 050)...

 Película encontrada: Pelicula 050 (1989) - Director X
Comparaciones en búsqueda: 1
Pasos del algoritmo:
Comparando 'Pelicula 050' en posición 49
¡Elemento 'Pelicula 050' encontrado en posición 49!

 RESUMEN:
Se ordenó con el método QuickSort porque la lista es muy grande (100 elementos),
lo cual lo hace más eficiente que otros algoritmos como Bubble Sort.
Se realizó la búsqueda con el método binario porque es más rápido que la búsqueda lineal en listas ordenadas.

 Tiempo de ordenamiento: 0.000000 segundos
 Tiempo de búsqueda: 0.000000 segundos
```

- Se verificó la salida del ordenamiento (orden correcta y cantidad de pasos).
- La búsqueda binaria encontró exitosamente "Pelicula 050" y mostró el recorrido paso a paso.
- No se presentaron errores en tiempo de ejecución y los resultados fueron coherentes.

¿Por qué se usó QuickSort para ordenar?

Este es un algoritmo muy eficiente para listas grandes como la del ejemplo (100 elementos). Tiene una complejidad promedio de **O(n log n)**, es más rápido que Bubble Sort y no requiere memoria adicional como Merge Sort. Por eso es ideal para ordenar rápidamente listas extensas.

¿Por qué se usó búsqueda binaria?

Dado que la lista está ordenada y contiene muchos elementos, la búsqueda binaria es más eficiente que la búsqueda lineal, con una complejidad de **O(log n)** en lugar de **O(n)**. Además, en el contexto del ejemplo, garantiza un rendimiento óptimo y demostrable.

Este caso práctico complementa el resto del trabajo, permite justificar la elección de métodos en función del contexto, cumpliendo así con un análisis completo del rendimiento y la lógica de programación aplicada.

4 . Metodología Utilizada

Antes de comenzar con el proyecto se llevó a cabo un exhaustivo análisis del material de apoyo que proporciona la catedra, en base a ello se opto por este tema y comenzamos a buscar bibliografía sobre el tema de algoritmos de búsqueda y ordenamiento, principalmente en videos de youtube para comprender los métodos como Bubble Sort, Merge Sort, QuickSort, búsqueda lineal y binaria, sitios como GeeksforGeeks que brindan conceptos y referencias muy útiles para avanzar. También se utilizó chatgpt como herramienta de apoyo.

Para comenzar se armó una versión básica del menú para ingresar películas manualmente. Luego se fue agregando los métodos de ordenamiento y búsqueda. Se probo cada función por separado y, una vez que todo funcionaba, se sumó el ejemplo especial con una lista de 100 películas precargadas.

Además, se incluyó una medición del tiempo de ejecución para poder analizar el rendimiento de QuickSort y la búsqueda binaria en un caso con muchos datos.

Aunque el código parecía estar completo, se identificó un error relacionado con la validación de entradas del usuario..:

Esto ocurría porque el código estaba diseñado para que el usuario introduzca números, pero al no hacerlo se paraba y presentaba un error sin avisar el problema o dar otra opción

```
4. Guardar último resultado de ordenación
0. Salir
Elige una opción: 1
¿Cuántas películas quieres ingresar? moana
Traceback (most recent call last):
  File "c:\Users\54379\OneDrive\Desktop\tecnicatura\Programacion1-tp-final\busqueda_ordenamiento_python_v2\main.py", line 158, in <module>
    menu()
  File "c:\Users\54379\OneDrive\Desktop\tecnicatura\Programacion1-tp-final\busqueda_ordenamiento_python_v2\main.py", line 75, in menu
    lista = pedir_lista_peliculas()
            ^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\54379\OneDrive\Desktop\tecnicatura\Programacion1-tp-final\busqueda_ordenamiento_python_v2\peliculas_utils.py", line 6, in pedir_lista_pelicula
    n = int(input("¿Cuántas películas quieres ingresar? "))
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'moana'
PS C:\Users\54379>
```

El código original fue este:

```python
def pedir_lista_peliculas():
    lista = []
    n = int(input("¿Cuántas películas quieres ingresar? "))
    for i in range(n):
        print(f"\nPelícula {i+1}:")
        titulo = input("  Título: ")
        anio = int(input("  Año: "))
        director = input("  Director: ")
        lista.append({"titulo": titulo, "anio": anio, "director": director})
    return lista
```

Se soluciono esto modificando el código

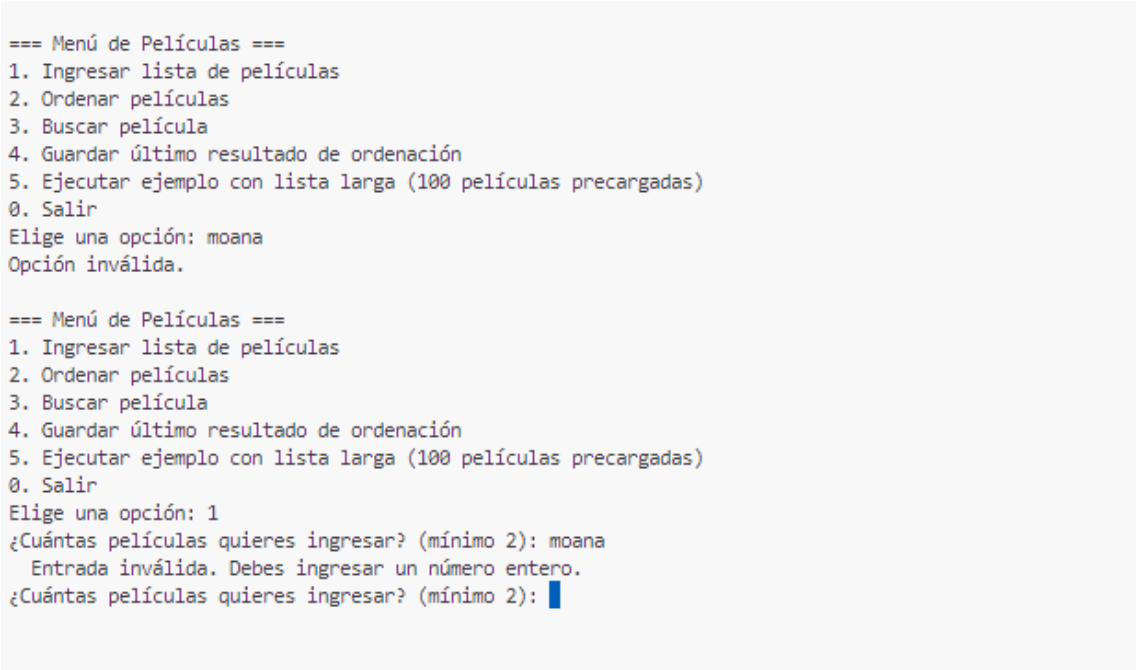
```python
def pedir_lista_peliculas():
    while True:
        entrada = input("¿Cuántas películas quieres ingresar? (mínimo 2): ")
        try:
            n = int(entrada)
            if n < 2:
                print(" Debes ingresar al menos 2 películas para continuar con el proceso de ordenarlas o buscarlas.")
                continue
            break
        except ValueError:
            print("  Entrada inválida. Debes ingresar un número entero.")

    lista = []
    for i in range(n):
        print(f"\nPelícula {i+1}:")
        titulo = input("  Título: ")

        while True:
            anio_str = input("  Año: ")
            if anio_str.isdigit():
                anio = int(anio_str)
                break
            else:
                print(" El año debe ser un número entero.")

        director = input("  Director: ")
        lista.append({"titulo": titulo, "anio": anio, "director": director})
    return lista
```

Ante la introducción de texto no válido en el menú, el código responde de manera controlada mediante una excepción



Herramientas y recursos utilizados

- Visual Studio Code.
- Python 3.11.
- Librerías: time para medir duración.
- GitHub para guardar los avances y compartir el trabajo.

Trabajo colaborativo

El trabajo fue realizado de forma colaborativa, distribuyendo las tareas de la siguiente manera:

- Elección del tema, implementación y búsqueda de películas en conjunto.
- Un miembro se encargó del diseño base del main y otro del utils, y READMI,
- Ambos para la implementación y mejora del código, la comunicación fue por mensaje, con sugerencias y posibles ideas de mejoras.
- Trabajo conjunto para integrar el ejemplo con 100 películas y explicar por qué se optó por QuickSort y búsqueda binaria.
- También fue conjunta la redacción del informe y la carga en GitHub, subiendo cada miembro la parte de la carpeta que tenía terminada.

5. Resultados Obtenidos

El programa permite ejecutar correctamente los algoritmos implementados y guardar los resultados. Entre los logros obtenidos se destacan:

- Funcionamiento exitoso de todos los métodos de ordenamiento y búsqueda.
- Conteo y visualización de pasos, comparaciones e intercambios.
- Claridad en el menú interactivo.
- Posibilidad de ver un ejemplo claro con 100 elementos, visibilidad del tiempo de respuesta al utilizar los métodos mas eficientes.

El código del proyecto se encuentra disponible en el repositorio colaborativo y publico: https://github.com/Anne-col/Trabajo-Integrador-Programacion-I.git

6. Conclusiones

Este trabajo permitió no solo aplicar conceptos aprendidos en clase, sino también enfrentar un proceso real de desarrollo: desde la investigación y el diseño, la colaboración y la conciliación en equipo. Además, aprender a programar diferentes algoritmos de búsqueda y ordenamiento, a entender cuándo conviene usar cada uno y a tomar decisiones técnicas justificadas. Poner en práctica la comparación entre métodos es clave para desarrollar criterio técnico por parte de los estudiantes.

Incorporar una lista larga de 100 películas precargadas, dio la posibilidad de ver en acción cómo QuickSort y búsqueda binaria resuelven problemas de forma ágil y precisa.

Como mejoras futuras, se podría conectar la lista de películas con un archivo externo (como una base de datos o archivo CSV).

En cuanto a dificultades, la principal fue manejar errores de entrada del usuario, lo cual se pudo resolver mediante el trabajo colaborativo, mediante una retroalimentación constante, lo cual fue muy enriquecedor.

Por último, pero no menos importante, favorece la distribución organizada de tareas, una comunicación activa y abierta, y lo que es aún mejor, lograr un complemento de habilidades. Por todo ello el trabajo colaborativo permite llegar a una versión completa y funcional del proyecto.

7. Bibliografía

- Python Software Foundation. (2024). Python 3 Documentation. https://docs.python.org/3/
- GeeksforGeeks. (2024). Sorting Algorithms Explained. https://www.geeksforgeeks.org
- ChatGPT, OpenAI. (2025). Asistencia en redacción y depuración de código.
- Notas de Catedra programación I UTN