Trabajo Integrador - Cátedra de Programación I

Algoritmos de Búsqueda y Ordenamiento con Análisis de Eficiencia en Python

• **Alumnos**:

Buchek Lautaro – lautaro.buchek@tupad.utn.edu.ar

Castellini Gonzalo – gonzalo.castellini@tupad.utn.edu.ar

• **Materia**: Programación I

• **Profesor/a**: Cinthia Rigoni

• **Tutor**: Martín A. García

Link a repositorio :

https://github.com/lBuKi19/TRABAJO-INTEGRADOR/tree/main

Link a video:

https://drive.google.com/file/d/1xsZmYnsSB-93hKgAiHjA6XweE1Mul4Qe/view?usp=share_link

• **Fecha de Entrega**: 09/06/2025

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

ÍNDICE

Introducción

Los algoritmos de búsqueda y ordenamiento constituyen uno de los pilares fundamentales de la programación, ya que permiten organizar y recuperar información de manera eficiente en diversos contextos computacionales. Este trabajo se enfoca en el estudio, implementación y análisis comparativo de estos algoritmos, debido a su relevancia tanto en aplicaciones académicas como en sistemas del mundo real, tales como motores de búsqueda, bases de datos y procesamiento de grandes volúmenes de datos.

La elección de este tema se justifica por su importancia en la optimización de recursos computacionales. En programación, la eficiencia de un algoritmo puede determinar la viabilidad de una solución, especialmente cuando se trabaja con conjuntos de datos extensos. Comprender cómo funcionan estos métodos, sus ventajas, desventajas y casos de uso permite a los desarrolladores seleccionar la estrategia más adecuada según el problema a resolver.

El objetivo principal de este trabajo es implementar algoritmos clásicos de ordenamiento (como Bubble Sort, Selection Sort, Insertion Sort, Merge Sort y Quick Sort) y de búsqueda (Lineal y Binaria), analizando su eficiencia mediante notación Big O y pruebas empíricas. Además, se busca comparar su rendimiento en diferentes escenarios, identificando cuándo es preferible utilizar uno u otro.

A través de este análisis, se espera demostrar la importancia de elegir algoritmos adecuados para cada situación, así como reforzar los conceptos teóricos mediante su aplicación práctica en Python. Los resultados obtenidos servirán como referencia para futuros desarrollos donde la optimización del tiempo y espacio de ejecución sea un factor crítico.

Marco Teórico

Antes de embarcarnos en la explicación y desarrollo del caso aplicado en el presente trabajo, es preciso enmarcar teóricamente los conceptos que luego se implementarán en el programa.

Por un lado se encuentra el concepto de ordenamiento, que implica la correcta organización de un conjunto de datos de acuerdo a un criterio específico. Este criterio dependerá de las características particulares de aquello que deseamos ordenar. Por ejemplo, en caso de estar tratando con una lista de contactos de teléfono, podemos requerir ordenarlos alfabéticamente.

Tanto para el ordenamiento como para la búsqueda, se encuentran diferentes algoritmos que, si bien tienen el mismo fin, cada uno de ellos cuenta con sus respectivas ventajas y desventajas. Para comprender mejor la eficiencia de estos algoritmos, se utiliza la métrica Big O(n), que representa el tiempo de ejecución en relación a la entrada de datos proporcionada.[1]

Entre los algoritmos de ordenamiento más comunes se encuentran:

Bubble sort

El algoritmo burbuja por su método de comparación de elementos adyacentes. Este algoritmo recorre todos los elementos del conjunto de datos de a pares, y los intercambia en caso de que estén desordenados según el criterio especificado. Su implementación y comprensión es sencilla, pero su procedimiento lo vuelve lento e ineficiente cuando la cantidad de datos a procesar es grande.

Selection sort

El algoritmo por selección funciona identificando al elemento más pequeño/grande de la lista, y ubicándolo en la primera posición, ya que de este modo, en el próximo recorrido, no será necesario pasar por este elemento debido a que no habrá ningún otro que sea menor/ mayor que él. Al igual que el ordenamiento por burbuja, este algoritmo es simple pero poco eficiente cuando el conjunto de datos es elevado.

[1] Tabla comparativa de eficiencia, ubicada al final del marco teórico. Página 5

Quick sort

El algoritmo de ordenamiento rápido funciona mediante la estrategia “divide y conquista”. Primero se selecciona un elemento a modo de “pivote” de manera arbitraria (puede ser primer/último elemento, etc) el cual funciona para particionar la lista en los elementos menores a él (a su izquierda) y aquellos mayores (a su derecha). Este procedimiento se realiza de manera recursiva, con las sublistas formadas alrededor del pivote hasta que cada una de ellas queda con un solo elemento. El ordenamiento rápido será eficiente para conjuntos grande datos, dependiendo también de la elección que se realice del pivote.

Insertion Sort

El algoritmo por inserción, funciona tomando los elementos de la lista desordenada, y colocándolos en su sitio correcto en la porción ordenada de la lista(se van colocando los elementos al inicio). A medida que el procedimiento se ejecuta, la porción de lista ordenada va a ir aumentando en tamaño, incorporando los elementos de la porción desordenada hasta que ésta quede vacía. Este algoritmo, tiene la particularidad de ser estable, ya que sus tiempos de ejecución promedio serán iguales a aquellas situaciones de peor caso posible, así como también ser eficiente cuando se lo aplica a conjuntos casi ordenados. Pero será ineficiente frente a conjuntos grandes de datos debido a su complejidad de tiempo cuadrática.

Algoritmos de Búsqueda

Los algoritmos de búsqueda emplean diferentes procedimientos con el fin de buscar elemento/s que coincidan con un criterio específico. Su implementación y eficiencia, dependerá de las características propias del conjunto de datos con el cual se esté trabajando, así como el criterio de búsqueda que se quiera utilizar.

Búsqueda lineal

Este algoritmo recorre cada elemento de la lista de manera secuencial y lo compara con el criterio de búsqueda especificado para encontrar coincidencias. La búsqueda lineal no requiere que el conjunto de datos esté ordenado y es flexible, ya que puede utilizarse con diferentes tipos de datos. Debido a sus características, es ineficiente para grandes listas, ya que puede ocurrir que el elemento que estemos

buscando se encuentre en la última posición, y por lo tanto se tenga que recorrer todo el conjunto de elementos de forma secuencial.

Búsqueda binaria

Este algoritmo eficiente tiene como requisito excluyente que la lista se encuentre ordenada en función del criterio de búsqueda. Por ejemplo, si vamos a buscar una persona en una lista de contactos de teléfono, la misma debe estar ordenada alfabéticamente.

El procedimiento que implementa este algoritmo, implica dividir de manera iterativa en mitades la lista y analizar los elementos de los extremos de las mitades con aquel que se busca. Si el elemento es mayor o menor, se puede descartar la mitad correspondiente (debido a que ya está ordenada), y de este modo ir achicando la lista hasta encontrar coincidencias en caso de que haya.

Tabla comparativa de eficiencia (notación Big O(n))		
Tipo	**Mejor caso**	**Peor caso**
Bubble sort	O(n)	O(n²)
Selection sort	O(n)	O(n²)
Insertion sort	O(n)	O(n²)
Quick sort	O(n log n)	O(n²)
Búsqueda lineal	O(1) (primer elemento)	O(n)
Búsqueda binaria	O(1) (elemento de la mitad de la lista)	O(log n)

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Caso Práctico: Análisis y Ordenamiento de Películas y Series IMDB

Descripción del problema a resolver

El objetivo principal de este trabajo es construir un sistema en Python capaz de **leer, analizar, ordenar y buscar datos** de películas y series provenientes de un archivo con extensión `.csv`.[2]Este archivo contiene información relevante como título, año, género, rating y cantidad de votos. A partir de estos datos, el sistema permite aplicar distintos **algoritmos de ordenamiento y búsqueda**, con el fin de:

- Visualizar películas/series ordenadas por distintos criterios.
- Visualizar mediciones de eficiencia en cada prueba para fines prácticos.
- Buscar películas/series por título, género o año.

Explicación de decisiones de diseño

1. **Estructura modular**:
   Cada funcionalidad está encapsulada en funciones, mejorando la legibilidad y el mantenimiento del código. La aplicación del trabajo consta de tres archivos .py:
   - *main.py* : Donde se aloja la principal función dedicada a la interfaz e interacción del usuario para las diferentes opciones de programa.
   - *funciones_algoritmos.py* : Donde se alojan las respectivas funciones para los algoritmos de búsqueda, ordenamiento y recomendación.
   - *formateo_visualzacion.py* : Dedicado a las funciones de estandarización y verificación de los datos del archivo .csv ingresado, y correcta visualización de datos arrojados.

[2] Para observar las columnas del csv trabajas dirigirse al Anexo II. Página 25

2. **Validación y limpieza de datos**:

   Durante la carga del archivo `IMBDcopy.csv`, los campos como el año, el rating y los votos son limpiados y transformados para evitar errores de formato. Esto asegura que los algoritmos puedan ejecutarse correctamente sobre datos consistentes.[3]

3. **Elección de algoritmos de ordenamiento**:

   - **Bubble Sort** : Elegido por su simplicidad para propósitos didácticos y comparativa con otros algoritmos. Versión lenta para comparar eficiencia.

   - **Quick Sort** : Seleccionado por su gran eficiencia en tiempo promedio (`O(n log n)`), y utilidad con grandes volúmenes de datos, como es el caso.

4. **Búsqueda lineal y binaria**:

   - Se emplea **búsqueda lineal** para texto (título, género y año) por su flexibilidad en cadenas, incluso cuando el input del usuario no coincida de manera perfecta con el título de película/serie que se desea buscar.

   - Se aplica **búsqueda binaria** en años y rating, aprovechando que los datos pueden ser fácilmente ordenados numéricamente.

5. **Interfaz tipo consola**:

   - El menú interactivo guía al usuario para navegar fácilmente entre visualización, estadísticas, ordenamientos y búsquedas.

[3] Estos errores se detallan en el apartado de Detalle de funciones implementadas. Página 11

Detalle de funciones implementadas

1. **Carga y validación de datos (*cargar_peliculas*)**
2. **Visualización formateada (*mostrar_peliculas*)**
3. **Validacion de input de usuario (*validacion*)**
4. **Algoritmos de ordenamiento**
    - *bubble_sort*
    - *quick_sort*
5. **Algoritmos de búsqueda**
    - *busqueda_lineal*
    - *busqueda_binaria*
6. **Algoritmo de recomendación (**algo_recomendaciones**)**
7. **Menú interactivo (`main`)**

DESCRIPCIÓN DE FUNCIONES IMPLEMENTADAS

Funciones de Búsqueda y Ordenamiento

Aquellas contenidas en *funciones_algoritmos.py*

Para la implementación de los algoritmos utilizados en el trabajo, se tomaron como base los códigos proporcionados por la cátedra, a los cuales se aplicaron las modificaciones pertinentes para poder adecuarlos a la aplicación del archivo .csv, así como también algunas funcionalidades extras.

Búsqueda y Ordenamiento según categoría

Se modificaron los algoritmos para que las funciones tomen parámetros adicionales, como es el caso de las diferentes categorías del archivo .csv (año, rating, género), para que el usuario pueda ejecutar dicho algoritmo de manera flexible, dependiendo de sus necesidades.

En este caso la función *busqueda_lineal,* toma 3 parámetros*:*

- películas = lista correctamente formateada del archivo .csv

- input = la búsqueda específica (ej, si es de género, podría ser drama)
- opción = la categoría elegida por el usuario (género, título, año)

```python
def busqueda_lineal(peliculas, input, opcion):
    """Busca películas por título/genero usando búsqueda lineal"""
    encontradas = []
    columna = opcion
    if opcion == 'titulo':
        print(f"🔍 Buscando películas con '{input}' en el título...")
    elif opcion == 'genero':
        print(f"🔍 Buscando películas de genero '{input}'...")
    elif opcion == 'año':
        print(f"🔍 Buscando películas por año '{input}'...")

    inicio = time.time()

    # Búsqueda lineal - revisar cada elemento
    for pelicula in peliculas:
        if opcion == 'año':
            if input == pelicula[columna]:
                encontradas.append(pelicula)
        else:
            if input in pelicula[columna].lower():
                encontradas.append(pelicula)

    tiempo = time.time() - inicio
    print(f"⏱️  Tiempo de ejecución: {tiempo:.6f} segundos")

    if encontradas:
        print(f"🔍 Se encontraron {len(encontradas)} resultados")
        return encontradas
    else:
        print(f"❌ No se encontraron películas o series con esas características")
```

Implementación flexible en función de inputs del usuario.

Orden ascendente y descendente en Quick sort

Se añadió funcionalidad de orden ascendente o descendente en función *quick_sort* mediante la adición de parámetro reverse , el cual es un booleano que cambia de valor dependiendo si seleccionamos ascendente (True) o descendente (False, que viene por defecto). Esta variable, impacta en un condicional dentro de la función que cambia el orden de sumatoria de las mitades(left y right).

```python
def quick_sort(peliculas, input, reverse=False):
    """Ordena películas según categoría seleccionada usando Quick Sort"""

    def quicksort(arr):
        """Función recursiva de Quick Sort"""
        if len(arr) <= 1:  # Caso base
            return arr

        pivot = arr[len(arr) // 2][input]  # Pivote del medio
        right = [x for x in arr if x[input] > pivot]    # Mayores que pivote
        middle = [x for x in arr if x[input] == pivot] # Iguales al pivote
        left = [x for x in arr if x[input] < pivot]   # Menores que pivote

        #condicional que determina orden ascendente o descendente
        if reverse:
            return quicksort(left) + middle + quicksort(right)
        else:
            return quicksort(right) + middle + quicksort(left)  # Combinar r
```

Busqueda binaria y algunas particularidades

Debido a que el algoritmo de búsqueda binaria requiere que el conjunto en el cual se aplica esté ordenado, realizamos una ordenamiento inicial de los datos con la función *sorted()* incluída en python, para luego poder ejecutar la búsqueda binaria. Si bien esta implementación cumplía con nuestros requerimientos, y ordenaba de manera correcta los datos, decidimos implementar la función de *quick_sort()* que habíamos definido en el programa previamente, y de esta manera poder abordar el algoritmo con una solución más adecuada al trabajo práctico en cuestión.

```python
def busqueda_binaria(peliculas, input_categoria, eleccion):
    """Busca segun categoria seleccionada usando búsqueda binaria"""

    print(f"📅 Buscando películas/series usando búsqueda binaria...")
    inicio = time.time()

    # Primero ordenar usando funcion ya definida (requisito para búsqueda binaria)
    pelis_ordenadas = quick_sort(peliculas, input_categoria, reverse=True)
    encontradas = []
    low, high = 0, len(pelis_ordenadas) - 1
```

Llamado de función quick_sort dentro de la búsqueda binaria.

Por curiosidad, decidimos probar con métricas de eficiencia y hallamos que la función *sorted()* incluída con python era más rápida que la función implementada por nosotros de *quick_sort()*. Este resultado, nos llevó a investigar sobre qué algoritmo está aplicando python para la función *sorted()*, el cual se denomina "timsort". Este algoritmo es una combinación del insertion sort y merge sort, y es el algoritmo predeterminado del lenguaje Python y Java.

Algoritmo de recomendación

A modo de integración de los conceptos y las funciones ya implementadas, realizamos un sencillo algoritmo de recomendación basado en géneros que ingrese el usuario. El algoritmo busca coincidencias (2 o 3 si es posible) de los géneros ingresados para poder arrojar los mejores resultados filtrados por rating. El usuario tiene la posibilidad de ingresar hasta tres géneros que sean de su interés.

```
def algo_recomendaciones(peliculas, input_generos):
    pelis_ordenadas = quick_sort(peliculas, input='rating')
    generos = input_generos.split()
    #armar lista de recomendaciones
    recomendaciones = []
    # buscar coincidencias de genero 2 o 3 si es posible
    coincidencias = 0
    for peli in pelis_ordenadas:
        for genero in generos:
            if genero in peli['genero'].lower():
                coincidencias +=1
        if coincidencias >= len(generos)-1:
            recomendaciones.append(peli)
        coincidencias = 0

    return recomendaciones
```

Se utiliza una lista ya ordenada por rating(llamando a quick_sort), y de este modo, las primeras ocurrencias que se ingresen a la lista de recomendaciones, serán las de mejor rating.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Funciones de formateo, visualización y validación

Aquellas contenidas en *formateo_visualizacion.py*

En un primer momento comenzamos a implementarlas mediante pandas, pero tuvimos problemas para familiarizarnos con la librería y la correcta lectura y formateo de datos del archivo .csv. De este modo, realizamos la aplicación práctica con la librería csv.

Consulta a la inteligencia artificial (Deepseek)

Para el correcto formateo y visualización de los datos, realizamos una consulta a DeepSeek ya que estábamos teniendo dificultades con la variación de la consistencia de los datos que se encontraban en las columnas del archivo. Estos problemas, se debían a que en campos como el año, aparecían diferencias en cuanto al formato de fechas que estaban registradas, así como también en los registros de rating.

year
(2018–)
(2016–)
(2015–2022)
-2022
(2022–)
(2022–)
(2013–)
(2008–2013)
(2022–)
-2022
(2016–)
(II) (2022)

```python
for fila in lector:
    try:
        # Procesamiento del año - extraer solo los 4 dígitos numéricos
        year_str = str(fila.get('year', '')).strip()
        if year_str.isdigit() and len(year_str) == 4:
            fila['year'] = int(year_str)
        else:
            # Extraer dígitos de cadenas como "(1994)" o "1994-01-01"
            digits = ''.join([c for c in year_str if c.isdigit()])
            fila['year'] = int(digits[:4]) if len(digits) >= 4 else 0
```

Ejemplo distintos formatos de año dentro del csv y formateo en función.

Esto se debe en parte, a las series que todavía siguen en funcionamiento y no tienen años de finalización por ejemplo.

La inteligencia artificial nos proporcionó una función para poder manejar estos casos y formatearlos de manera correcta. Ésta funciona comprobando aquellos caracteres solo números con *isdigit()*, así como también que la duración del string sea correcta (4 números) antes de guardar el dato en lista.

Visualización de datos

Implementamos una función para poder visualizar los resultados de las búsquedas de manera clara, ya que de lo contrario estos se muestran en una lista, con todos los campos de corrido, lo que dificulta su comprensión. Utilizamos un loop con la función incluída de python enumerate() para recorrer la lista de resultados.

```
================================================================================
Mostrando 10 de 9957 películas:
================================================================================
#   TÍTULO                                   AÑO    GÉNERO              RATING
================================================================================
1   BoJack Horseman                          2014   Animation, Comedy...  9.9
2   1899                                     2022   Drama, History, H...  9.6
3   Avatar: The Last Airbender               2005   Animation, Action...  9.6
4   Dexter                                   2006   Crime, Drama, Mys...  9.6
5   JoJo's Bizarre Adventure                 2012   Animation, Action...  9.6
6   Avatar: The Last Airbender               2005   Animation, Action...  9.6
7   Stranger Things                          2016   Drama, Fantasy, H...  9.6
8   Breaking Bad                             2008   Crime, Drama, Thr...  9.5
9   Avatar: The Last Airbender               2005   Animation, Action...  9.5
10  Dark                                     2017   Crime, Drama, Mys...  9.5
```

Visualización de datos al buscar por rating a modo de tabla.

Manejo de datos faltantes, como el caso de películas o series sin rating, reemplazamos para que se muestren “N/A” y no arroje error.

```python
def mostrar_peliculas(peliculas, limite=10):
    barra_grande= '='*110
    """Mostrar películas en formato de tabla"""
    if not peliculas:
        print("No hay películas para mostrar.")
        return
    # Encabezado de la tabla
    print(f"\n{barra_grande}")
    print(f"Mostrando {min(len(peliculas), limite)} de {len(peliculas)} películas:")
    print(f"{barra_grande}")
    print(f"{'#':<3} {'TÍTULO':<40} {'AÑO':<6} {'GÉNERO':<18} {'RATING':<8}")
    print(barra_grande)

    # Mostrar cada elemento alineado
    for i, p in enumerate(peliculas[:limite]):
        # Acortar campos largos para mejor visualización
        titulo = p['title'][:39] + '...' if len(p['title']) > 39 else p['title']
        año = str(p['year']) if p['year'] > 0 else 'N/A'
        genero = p['genre'][:17] + '...' if len(p['genre']) > 17 else p['genre']
        rating = f"{p['rating']:.1f}" if p['rating'] > 0 else 'N/A'

        # Imprimir fila formateada
        print(f"{i+1:<3} {titulo:<40} {año:<6} {genero:<18} {rating:<8}")
```

Validación de input de usuario

Función implementada con el fin de validar datos de input del usuario como año o rating, y de este modo evitar la repetición de mismos bloques de código que se iban a aplicar en funciones de algoritmos diferentes. Simple realización mediante bucles *while* y verificación de valor de datos con *isdigit()*.

```python
def validacion(input_categoria):
    if input_categoria == 'año':
                while True:
                    eleccion = input("Ingrese el año a buscar: ").strip()
                    if eleccion.isdigit():
                        eleccion = int(eleccion)
                        break
                    else:
                        print("❌ Por favor ingrese un año válido (número entero).")
    elif input_categoria == 'rating':
                try:
                    eleccion = float(input("Ingrese el rating: "))
                except ValueError:
                    print("❌ Por favor ingrese un rating válido(número entero o con punto).")

    return eleccion
```

Función main() (contenida en *main.py*)

Esta función contiene el algoritmo de interfaz del programa, junto con el menú de opciones a seleccionar. Se utilizó una lista inicial para guardar aquellas opciones del programa y luego simplemente iterar sobre ellas con un loop para poder mostrarlas por pantalla al usuario,

```
lista_menu = ["\n1. Cantidad de Resultados por operación", "\nORDENAMIENTO:",
```

Por temas de espacio se muestra un recorte de la lista, esta contiene todas las opciones con las que luego iteramos para mostrar por pantalla.

```
# Loop por opciones de menu para mostrar
for item in lista_menu:
    print(item)
```

```
if opcion == "1":
    limite= input("¿Cuántas resultados deseas mostrar? (10 por defecto): ").strip()
    if limite.isdigit():
        limite = int(limite)
    else:
        limite = 10
    mostrar_peliculas(peliculas, limite)
elif opcion == "2":
    input_categoria = input("Ingrese la categoría que desea ordenar (rating, genero, año, t
    mostrar_peliculas(bubble_sort(peliculas, input_categoria))
elif opcion == "3":
    input_categoria = input("Ingrese la categoría que desea ordenar (rating, genero, año, t
    mostrar_peliculas(quick_sort(peliculas, input_categoria))
elif opcion == "4":
    input_titulo = input("Ingrese el título a buscar: ").strip().lower()
    mostrar_peliculas(busqueda_lineal(peliculas, input_titulo, opcion='titulo'))
elif opcion == "5":
    input_categoria = input("Ingrese el genero a buscar: ").strip().lower()
    mostrar_peliculas(busqueda_lineal(peliculas, input_categoria, opcion='genero'))
elif opcion == "6":
    input_categoria = validacion('año')
    mostrar_peliculas(busqueda_lineal(peliculas, input_categoria, opcion='año'))
elif opcion == "7":
    input_categoria = input("Ingrese la categoría que desea buscar por búsqueda binaria: ")
    eleccion_categoria = validacion(input_categoria)
    mostrar_peliculas(busqueda_binaria(peliculas, input_categoria, eleccion_categoria))
elif opcion == "8":
    input_generos = input("Ingrese los géneros que le interesan separados por espacios: ").
    mostrar_peliculas(algo_recomendaciones(peliculas, input_generos))
elif opcion == "9":
    print("¡Hasta luego! 👋")
    break
else:
    print("❌ Opción no válida. Por favor seleccione un número del 1 al 8.")
```

Por limitaciones de visualización del documento, se realizó un recorte de las opciones. Para observar el código completo dirigirse al repositorio en Github.[4]

Se implementó un bucle *while* con condicionales dentro para tratar los casos de opciones elegidas por usuario y en base a ellas, llamar las debidas funciones para su ejecución y luego mostrar los resultados por pantalla. Este bucle *while*, continúa iterando, para que el usuario pueda realizar cuantas operaciones seguidas desee, sin que el programa tenga que reiniciarse. El programa solo finaliza cuando se registra la opción 9.

Metodología Utilizada

El desarrollo de este proyecto se llevó a cabo en varias etapas, siguiendo un enfoque estructurado, progresivo e iterativo. A continuación, se detallan los pasos y recursos empleados:

Investigación previa

Antes de comenzar la codificación, se realizó una investigación bibliográfica y en línea sobre los siguientes temas:

- Algoritmos de ordenamiento clásicos: Bubble Sort, Selection Sort, Insertion Sort y Quick Sort.
- Algoritmos de búsqueda: búsqueda lineal y binaria.
- Estructuras de datos en Python (listas, diccionarios).
- Manejo de archivos CSV y normalización de datos con errores de formato, librerías como Pandas, csv.
- Fuentes utilizadas:

[4] El link al repositorio se encuentra en la bibliografía. Página 24

- Documentación oficial de Python: https://docs.python.org
- W3Schools Python tutorials, FreeCodeCamp.
- Material de la cátedra
- Stack Overflow (para resolución de errores puntuales)
- DeepSeek (inteligencia artificial)

Etapas de desarrollo

El desarrollo se dividió en fases específicas:

Fase 1: Búsqueda, Análisis y preparación del dataset

- Búsqueda de datasets y descarga en Kaggle.[5]
- Estudio del contenido de `IMBD.csv` para entender su estructura.
- Detección de errores comunes en los campos (fechas incompletas, decimales con comas, etc.).

Fase 2: Construcción del módulo de carga y validación

- Implementación de cargar_peliculas() con manejo de errores y limpieza de datos.

Fase 3: Diseño de visualización y validación

- Creación de mostrar_peliculas() para mostrar resultados en formato de tabla clara.
- Creación de validacion() para validar inputs del usuario.

Fase 4: Implementación de algoritmos

- Comprensión e implementación de código base de algoritmos provistos por la cátedra.

[5] Para visitar la web de Kaggle, dirigirse al apartado bibliográfico. Página 22.

- Adecuación de algoritmos para caso práctico particular y añadido de funcionalidades específicas.
- Cada algoritmo fue probado individualmente sobre subconjuntos de datos.

Fase 5: Menú interactivo y pruebas completas

- Integración de funciones en un menú amigable para el usuario.
- Prueba con lista simple como entrada de datos.
- Pruebas funcionales con distintas entradas, incluyendo casos límite y datos ausentes.

Fase 6: Modularización del programa en 3 partes

- main.py
- formateo_visualizacion.py
- funciones_algoritmos.py

Fase 7: Pruebas completas y análisis de resultados

- Búsqueda e implementación de un dataset.

Herramientas y recursos utilizados

- **Lenguaje de programación**: Python 3.13
- **IDE**: Visual Studio Code.
- **Librerías estándar**:
  - `csv`: lectura y parseo de archivos CSV
  - `time`: para medir rendimiento

  - `sys`: para manejo del programa

- **Control de versiones**:
  - Git (para seguimiento local de versiones)
  - GitHub (para trabajo colaborativo)
- **Sistemas operativos**: Windows 10 y MacOS

Resultados Obtenidos

Resultados generales logrados

- Se logró construir un sistema funcional en Python para analizar y gestionar datos de películas y series contenidas en un archivo `IMBD.csv`.
- Se implementaron correctamente 2 algoritmos de ordenamiento y 2 de búsqueda, y uno de recomendación, todos accesibles desde un menú interactivo.
- Se garantizó la limpieza y validación de los datos antes de procesarlos, corrigiendo automáticamente formatos inconsistentes en los campos de año y rating.
- La visualización de resultados se presenta en forma tabular clara y ordenada, permitiendo una interpretación intuitiva de la información.

Ejemplificación de casos de prueba realizados

Prueba	Descripción	Resultado
Carga de archivo CSV	Lectura de `IMBD.csv` con datos incompletos y diferentes formatos	✅ Filtrado y corrección exitosa
Ordenamiento por año	Usando Bubble Sort	✅ Orden descendente correcto
Ordenamiento por rating	Usando Quick Sort	✅ Resultados coincidentes, rápidos
Ordenamiento por género	Usando Quick Sort	✅ Orden alfabético funcional
Búsqueda por título y género	Entrada parcial (ej. "avengers")	✅ Coincidencias encontradas
Búsqueda por año	Usando Búsqueda Binaria	✅ Precisa y rápida tras ordenamiento

Evaluación de rendimiento

Para evaluar el rendimiento de los algoritmos, se midió el tiempo de ejecución con el módulo `time` sobre un conjunto de aproximadamente 10000 películas y series totales.

Algoritmo	Criterio	Tiempo promedio (seg.)	Observaciones
Bubble Sort	Año	12,146	Lento en listas grandes
Quick Sort	Año	0.011	Muy eficiente incluso con miles de registros
Búsqueda Lineal	Año (ej: 2024)	0.002	Muy eficiente en listas pequeñas (ej: 10 películas)
Búsqueda Binaria	Año (ej: 2024)	0.012	Muy eficiente incluso con miles de registros, pero ineficiente en listas pequeñas. (ej: 10 películas)

📌 *Quick Sort se recomienda para volúmenes grandes de datos debido a su excelente desempeño.*

Conclusión de los resultados

El sistema logró cumplir su propósito didáctico y funcional: aplicar algoritmos de programación sobre datos reales y ofrecer una interfaz que permite al usuario explorar, ordenar y buscar películas y series con facilidad. En cuanto a los

resultados empíricos de eficiencia medidos en unidad de tiempo, los datos obtenidos fueron congruentes con la investigación previa y la teoría de cada uno de los algoritmos, donde por ejemplo, la búsqueda de burbuja se hace muy lenta al aplicarlo a conjuntos grandes de datos y por lo tanto, su aplicabilidad a casos reales de uso se ve limitada.

El código fue robusto frente a entradas incompletas, y el rendimiento fue aceptable para un entorno académico.

Conclusiones Finales

Durante el desarrollo de este trabajo, se pudo afianzar y poner en práctica varios conceptos clave de la programación, como la lectura y procesamiento de archivos CSV, la implementación de algoritmos de ordenamiento y búsqueda, y el uso de estructuras de datos como listas y diccionarios. También se profundizó en el manejo de errores, la validación de datos y la medición de tiempos de ejecución para comparar la eficiencia de los algoritmos implementados.

El tema trabajado resultó de gran utilidad, no solo como ejercicio académico, sino también por su aplicabilidad en proyectos reales donde es necesario procesar y analizar grandes volúmenes de información. El manejo de datos estructurados, como los de una base de películas, es una habilidad transversal que puede aplicarse en sistemas de recomendación, motores de búsqueda, análisis estadísticos y muchas otras áreas del desarrollo de software.

Como posibles mejoras futuras, se propone implementar una interfaz gráfica para mejorar la interacción con el usuario, y agregar funcionalidades como el guardado de resultados en archivos separados y manejo de duplicados. A pesar de no haber podido implementar la librería pandas , sería interesante poder incorporar su uso para optimizar el manejo de datos.

Durante la realización del proyecto surgieron algunas dificultades, principalmente relacionadas con la limpieza de los datos del archivo CSV, como se ha detallado en el desarrollo del trabajo. Estas complicaciones, se resolvieron con la asistencia de la inteligencia artificial, herramienta que nos permitió comenzar a acumular

conocimiento en áreas que no estábamos familiarizados. Otra dificultad fue lograr que los algoritmos funcionen correctamente con diferentes tipos de datos (números, cadenas, listas), lo cual requirió una adaptación del código base y una aplicación propia destinada al uso específico que se requería.

En definitiva, el proyecto permitió aplicar de forma concreta y práctica muchos de los conocimientos adquiridos durante la cursada, y fue una experiencia muy enriquecedora en términos de trabajo colaborativo, resolución de problemas y consolidación de habilidades en programación.

BIBLIOGRAFÍA

-Búsqueda inicial de datasets para caso práctico:

https://www.kaggle.com/

https://www-datablist-com.translate.goog/learn/csv/download-sample-csv-files?_x_tr_hist=true

https://wsform-com.translate.goog/knowledgebase/sample-csv-files/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

-Sorting Techniques - Python 3.13 Documentation. Disponible en:

https://docs.python.org/3/howto/sorting.html

-Búsqueda y Ordenamiento en Programación. Universidad Tecnológica Nacional (2025)

-Algoritmos de ordenamiento explicados con ejemplos. Freecodecamp :

https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/

-Enumerate() function. w3schools. Disponible en:

https://www.w3schools.com/python/ref_func_enumerate.asp

-Timsort. Data Structures. geekforgeeks. Disponible en:

https://www.geeksforgeeks.org/timsort/

-Sistema de búsqueda iMDB. Medium. Disponible en:

https://medium.com/@pankhudi.g123/optimize-imdb-search-9e28ee701f8b

🌐 Repositorio de proyecto en GitHub

https://github.com/IBuKi19/TRABAJO-INTEGRADOR/tree/main

Anexos

Anexo I

Diccionario de datos de programa main.py			
Campo	***Tipo***	***Valores aceptados***	***Descripción***
barra	str	-	formateo de signos “=” seguidos para separar información al mostrar por pantalla
lista_menu	list	-	lista con opciones de selección que se muestran a usuario
limite	int	solo dígitos	cantidad de resultados a mostrar luego de cada acción (valor 10 por defecto)
cargar_peliculas	funcion	-	función para cargar datos de archivo .csv a lista
peliculas	list	-	variable para almacenar el resultado tras llamar la función *cargar_peliculas()*
opcion	int	solo dígitos 1 - 9	variable para almacenar input de opción de usuario

validacion	función	1 parámetro	función de validación de input de año y rating
input_categoria	str	'genero' 'año' 'rating'	variable para almacenar selección de categoría de usuario
bubble_sort	función	2 argumentos	algoritmo de ordenamiento burbuja
quick_sort	función	mínimo 2 argumentos máximo 3	algoritmo de ordenamiento rápido
busqueda_lineal	función	3 argumentos	algoritmo de búsqueda lineal
busqueda_binaria	función	3 argumentos	algoritmo de búsqueda binaria
eleccion	float o int dependiendo el caso	solo dígitos	variable para guardar año o rating que ingresa el usuario para búsqueda
mostrar_peliculas	función	1 argumento máximo 2	función para mostrar resultados en formato tabla
algo_recomendaciones	función	2 argumentos	algoritmo de recomendación basado en géneros y rating

Anexo II : Captura de archivo csv original con las columnas utilizadas en el trabajo

title	year	certificate	duration	genre	rating
Cobra Kai	(2018–)	TV-14	30 min	Action, Comedy, Drama	8.5
The Crown	(2016–)	TV-MA	58 min	Biography, Drama, History	8.7
Better Call Saul	(2015–2022)	TV-MA	46 min	Crime, Drama	8.9
Devil in Ohio	-2022	TV-MA	356 min	Drama, Horror, Mystery	5.9
Cyberpunk: Edgerunners	(2022–)	TV-MA	24 min	Animation, Action, Adventure	8.6
The Sandman	(2022–)	TV-MA	45 min	Drama, Fantasy, Horror	
Rick and Morty	(2013–)	TV-MA	23 min	Animation, Adventure, Comedy	9.2
Breaking Bad	(2008–2013)	TV-MA	49 min	Crime, Drama, Thriller	9.5
The Imperfects	(2022–)	TV-MA	45 min	Action, Adventure, Drama	6.3
Blonde	-2022	NC-17	166 min	Biography, Drama, Mystery	6.2
Stranger Things	(2016–)	TV-14	51 min	Drama, Fantasy, Horror	8.7
End of the Road	(II) (2022)	R	89 min	Action, Crime, Drama	4.7
The Walking Dead	(2010–2022)	TV-MA	44 min	Drama, Horror, Thriller	8.1

Anexo III

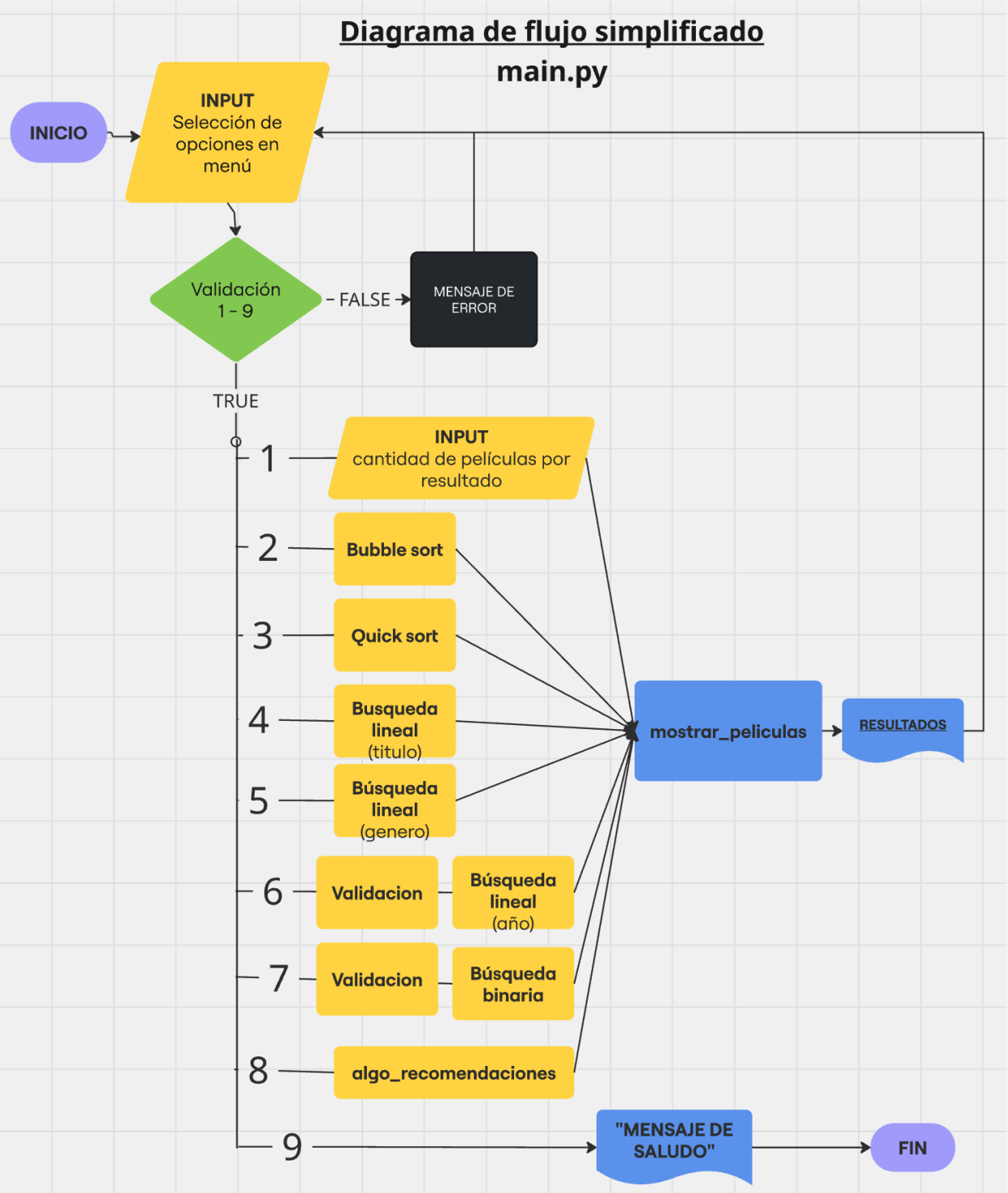

Diagrama de flujo realizado en Miró.