UTN

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

TRABAJO INTEGRADOR

PROGRAMACIÓN

COMISIÓN: 24

ALUMNOS:
FACUNDO ARRIETA
LUCIANO ANDRELO

BÚSQUEDA AND ORDENAMIENTO

PROFESOR: AUS BRUSELARIO, SEBASTIÁN
TUTOR: GUBIOTTI, FLOR

Plantilla para Trabajos Integradores – Programación I

Datos Generales

- Título del trabajo: BÚSQUEDA AND ORDENAMIENTO
- Alumnos: Facundo Arrieta/Correo: facundo.arrieta2506@gmail.com

  Luciano Andrelo/Correo: andreloluciano91@gmail.com
- Materia: Programación I
- Profesor/a: Profesor: Aus, Bruselario, Sebastián

  Tutor: Gubiotti, Flor
- Fecha de Entrega: (Formato: día/mes/año)

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

---

1. Introducción

Introducción

En la programación moderna, la eficiencia de la manipulación de datos es fundamental. Este trabajo se centra en los algoritmos de búsqueda y ordenamiento debido a la

importancia que tiene en el procesamiento de información, en un mundo donde la información sobreabunda y está en constante cambio. Su relevancia radica en la capacidad de optimizar el rendimiento (reduciendo tiempos de ejecución y consumo de recursos) y garantizar la organización estructurada de los datos, facilitando su acceso y análisis en sistemas complejos. Son herramientas multidisciplinarias, aplicables desde bases de datos hasta servicios de streaming.

Los objetivos de este proyecto son:

- Implementar y demostrar algoritmos de búsqueda (lineal, binaria) y ordenamiento (burbuja, selección, inserción, Quick Sort).
- Ilustrar sus diferencias y aplicaciones a través de un sistema de gestión de catálogo de contenido.
- Ofrecer una plataforma interactiva para experimentar directamente con estos algoritmos y su impacto en la organización de la información.

Con ello, se busca solidificar la comprensión práctica de estas fundamentales herramientas algorítmicas para el desarrollo de software eficiente y robusto.

---

2. Marco Teórico

¿Qué son las estructuras de datos?

Una estructura de datos es una forma organizada de almacenar y gestionar la información. Elegimos distintas estructuras dependiendo del tipo de datos que manejamos y del tipo de operaciones que queremos realizar sobre ellos.

Por ejemplo, si quisiéramos representar las relaciones familiares entre varias personas, podríamos usar un árbol genealógico como estructura de datos. Este tipo de organización nos permite visualizar fácilmente cómo se relacionan los miembros de una familia, y responder preguntas como "¿quién es la madre de mi madre?" de forma casi instantánea.

Las estructuras de datos permiten gestionar grandes volúmenes de información de manera eficiente, lo cual es crucial en áreas como bases de datos, sistemas de búsqueda en internet, análisis de datos y muchas otras aplicaciones informáticas.

Además, las estructuras de datos y los algoritmos están conectados. Una estructura de datos no es realmente útil si no se puede recorrer, buscar o modificar mediante algoritmos eficientes, y los algoritmos dependen de tener una buena estructura sobre la cual operar.

Estructura de Datos: Arreglos (Arrays)

Un array (listas en Python) es una estructura de datos utilizada para almacenar múltiples elementos en una sola colección. Es uno de los tipos de estructuras más usados por diversos algoritmos debido a su simplicidad y eficiencia para acceder a los datos mediante índices.

Los arrays son especialmente útiles cuando necesitamos almacenar y manipular conjuntos de datos que pueden ser accedidos de forma secuencial o aleatoria. Por ejemplo, un algoritmo puede ser usado para recorrer un array con el objetivo de encontrar el valor más bajo entre sus elementos.

¿Qué es el ordenamiento (Sorting)?

Uno de los usos más comunes de los arrays es su ordenamiento, se refiere a la reorganización de los elementos según un operador de comparación. Este operador permite establecer un orden lógico, como menor a mayor, mayor a menor, ordenar cadenas por longitud, o cualquier criterio personalizado.

¿Por qué son importantes los algoritmos de ordenamiento?

El ordenamiento es fundamental en la informática por múltiples razones. Uno de los beneficios principales es que reduce la complejidad de muchos problemas. Además, es un paso previo esencial para otras tareas como la búsqueda eficiente de datos, la optimización de bases de datos, la gestión de memoria o el entrenamiento de modelos de machine learning.

Algunas aplicaciones destacadas:

- Buscar rápidamente el k-ésimo elemento (el menor o el mayor): una vez que el array está ordenado, encontrar estos valores puede hacerse en tiempo constante O(1).
- Algoritmos de búsqueda: muchos algoritmos eficientes, como la búsqueda binaria o ternaria, requieren que los datos estén ordenados para funcionar correctamente.
- Gestión de bases de datos: ordenar los registros mejora el rendimiento de las consultas, especialmente si se utiliza un índice primario.
- Visualización de datos: los datos ordenados pueden graficarse o representarse visualmente de manera más clara y comprensible.
- Análisis de datos y estadísticas: facilita la detección de patrones, tendencias y valores atípicos (outliers), ayudando en áreas como finanzas, estadística y ciencia de datos.

Ventajas de utilizar algoritmos de ordenamiento

- Eficiencia: los datos ordenados permiten búsquedas y análisis más rápidos.
- Mejor rendimiento: muchas tareas computacionales se aceleran si los datos están ordenados previamente.
- Análisis más simple: facilita la identificación de patrones y relaciones.
- Reducción del uso de memoria: ayuda a eliminar duplicados de forma más eficiente.
- Mejor visualización: los gráficos y representaciones visuales se benefician del orden en los datos.

Desventajas de los algoritmos de ordenamiento

- Inserción costosa: si los datos deben mantenerse ordenados todo el tiempo, insertar nuevos elementos puede ser más lento, ya que implica reorganizar la

estructura.

- Selección del algoritmo adecuado: no hay una solución única. Cada algoritmo tiene ventajas y desventajas según el tamaño de los datos, su orden inicial, y los recursos del sistema.
- Alternativas como el hashing: para algunos problemas, como detectar duplicados o buscar pares con una suma específica, el hashing puede ser más eficiente que ordenar.

¿Por qué Ordenar?

Muchos especialistas en Ciencias de la Computación consideran el ordenamiento como el problema más fundamental en el estudio de los algoritmos. Las razones son:

1. Necesidad real en aplicaciones prácticas: Por ejemplo, un banco debe ordenar los cheques por número para generar los resúmenes mensuales.
2. Subrutina en otros algoritmos: muchos algoritmos más complejos (como gráficos, planificación o análisis de datos) utilizan el ordenamiento como paso previo.
3. Riqueza en técnicas algorítmicas: existen muchos algoritmos de ordenamiento que emplean técnicas como recursión, divide y vencerás, y estructuras auxiliares. Estudiarlos permite comprender herramientas aplicables en otros contextos.
4. Cotas teóricas claras: se puede demostrar matemáticamente el límite inferior de eficiencia para cualquier algoritmo de ordenamiento basado en comparaciones. Muchos algoritmos alcanzan esa eficiencia límite de forma óptima: O(n log n).
5. Importancia en ingeniería de software: el algoritmo de ordenamiento más rápido para un caso concreto puede depender de factores como:
    - Conocimiento previo sobre los datos.
    - Composición de los datos (si están casi ordenados, duplicados, etc.).
    - Jerarquía de memoria del sistema (caché, RAM, disco).
    - Restricciones del entorno de programación.

Por eso, implementar un ordenamiento eficiente muchas veces implica decisiones algorítmicas más que ajustes finos de código.

Tipos de Ordenamiento

En programación, existen distintos algoritmos de ordenamiento, cada uno con sus ventajas, desventajas y niveles de eficiencia. En este trabajo usaremos los siguientes:

1. Ordenamiento por Burbuja (Bubble Sort)

Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto, repitiendo este proceso varias veces. Es fácil de implementar, pero ineficiente en listas grandes.
Funcionamiento: Recorre el array muchas veces, haciendo que los valores mayores "suban" al final.
Complejidad: Mejor caso lista ya ordenada, el algoritmo detecta que no hubo cambios en una sola pasada, en este caso su complejidad es O(n). Peor caso: la lista está en orden inverso, en cada pasada debe hacer muchas comparaciones e intercambios, lo que le da complejidad de $O(n^2)$

2. Ordenamiento por Selección (Selection Sort)

Busca el valor mínimo y lo coloca al principio, repitiendo el proceso para el resto de la lista. Usa pocas asignaciones de memoria, pero es lento en listas grandes
Funcionamiento: Divide la lista entre ordenada y desordenada, moviendo el mínimo cada vez.
Complejidad: Siempre $O(n^2)$. Mejor y peor caso son iguales en cantidad de comparaciones, siempre recorre toda la lista para encontrar el mínimo, sin importar el orden.

3. Ordenamiento por Inserción (Insertion Sort)

Inserta cada elemento en su lugar correcto dentro de la parte ya ordenada. Es eficiente para listas pequeñas o casi ordenadas, pero no escala bien con muchos datos.
Funcionamiento: Compara el elemento actual hacia atrás hasta ubicarlo correctamente.
Complejidad: Mejor caso O(n) con lista ordenada o casi ordenada, se realizan pocas comparaciones. $O(n^2)$ en el peor caso, con lista ordenada de forma inversa, requiere más comparaciones y movimientos.

4. Ordenamiento Rápido (QuickSort)

Elige un pivote, divide los elementos en menores y mayores, y repite el proceso recursivamente. Es muy rápido y eficiente, pero depende de una buena elección del pivote.
Funcionamiento: Aplica "divide y vencerás" para ordenar por secciones.
Complejidad: El mejor caso es O(n log n), ocurre cuando el pivote divide el arreglo de forma equilibrada (mitad y mitad). El peor caso es O(n²), ocurre cuando la lista ya está ordenada o completamente inversa si se elige mal el pivote. su caos promedio tambi´n es O(n log n).

Introducción a la Búsqueda

Con frecuencia, trabajaremos con grandes cantidades de información almacenada en arrays. Una operación común en este contexto es determinar si un array contiene un valor igual a cierta clave de búsqueda.
Este proceso se conoce como búsqueda, y consiste en localizar un elemento particular dentro de un conjunto de datos.

Existen diferentes tipos de búsqueda, y cada uno tiene sus ventajas y desventajas según el contexto. Los dos más conocidos son:

Búsqueda Lineal

La búsqueda lineal compara cada elemento del arreglo con la clave de búsqueda, uno por uno, desde el inicio hasta el final.
Este proceso continúa hasta que:

- Se encuentra el elemento (coincide con la clave), o
- Se recorren todos los elementos sin encontrarlo.

Este método funciona bien para arreglos pequeños o no ordenados, ya que no requiere ningún tipo de estructura previa.
Sin embargo, para arreglos grandes, este enfoque puede resultar ineficiente.
Complejidad: Mejor caso O(1), el elemento está en la primera posición, Peor caso el elemento está al final o no está O(n), recorre toda la lista

Búsqueda Binaria

La búsqueda binaria es una técnica más eficiente, pero requiere que los datos estén previamente ordenados.

El algoritmo funciona de la siguiente manera:

1. Se compara la clave buscada (X) con el elemento central del arreglo.
2. Si coincide, se ha encontrado el valor.
3. Si no coincide:
    - Si X es menor que el elemento central, la búsqueda se repite en la mitad izquierda del arreglo.
    - Si X es mayor, se repite en la mitad derecha.

Este proceso se repite dividiendo el arreglo a la mitad cada vez, hasta encontrar el elemento o confirmar que no se encuentra en el arreglo.
Complejidad: Mejor caso O(1), el elemento buscado está justo en el medio, Peor caso O(log n), el elemento está en un extremo de la lista o no está

En general, el tamaño de la lista es un factor importante a tener en cuenta al elegir un algoritmo de búsqueda. Si la lista es pequeña, es probable que la búsqueda lineal sea más eficiente. Sin embargo, si la lista es grande, es probable que la búsqueda binaria sea más eficiente.

¿Qué es notación O y por qué importa?

Notación O es una forma de analizar cómo crece el tiempo de ejecución de un algoritmo a medida que la entrada se hace más grande. Nos permite comparar algoritmos sin necesidad de ejecutarlos, útil para comparar diferentes algoritmos y elegir el más eficiente para una tarea determinada. Analiza cuánto tarda un algoritmo en función del tamaño de entrada.

Notación asintótica

Notación O describe el comportamiento en el peor de los casos, para tener una idea del límite superior del tiempo que puede tardar un algoritmo.

Ejemplo

Imaginá que llevás libros a un amigo:

- Método 1: Vas en auto, no importa si llevás 1 o 20 libros → siempre tardás 5 minutos.
  Esto es O(1) → tiempo constante.
- Método 2: Vas corriendo de a un libro por vez. Si llevás 2 libros, tardás 6 minutos; con 3 libros, 9 minutos.
  Esto es O(n) → tiempo lineal (crece con la cantidad de libros).

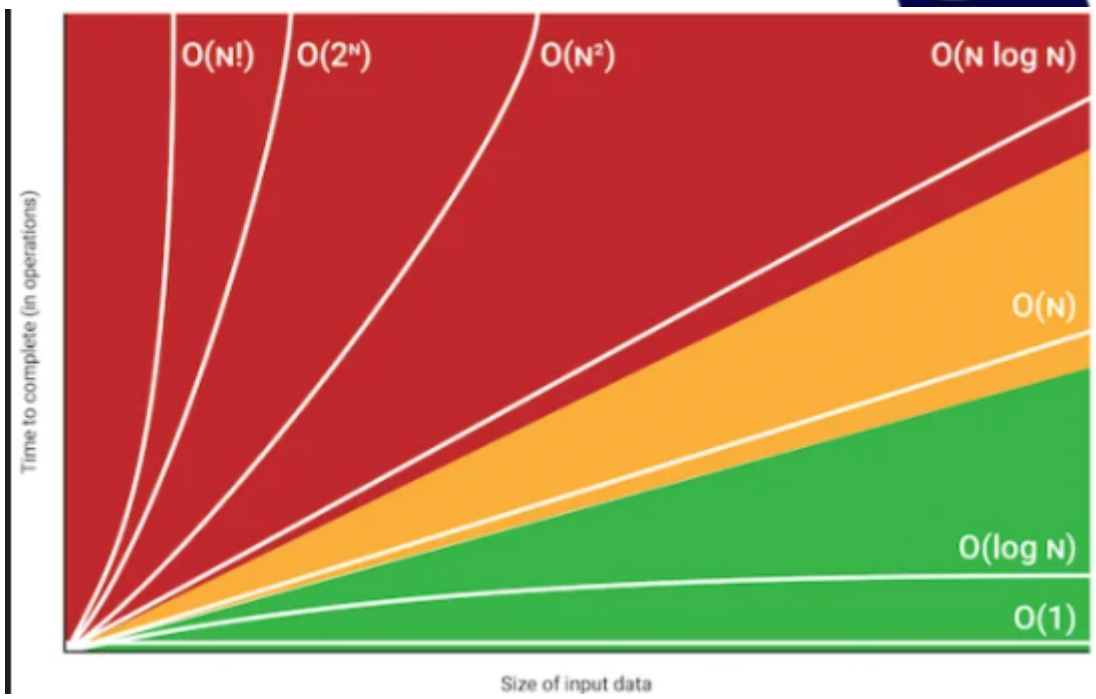
Los algoritmos de búsqueda lineal tienen un tiempo de ejecución de O(n), lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista. Esto significa que si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado.

Los algoritmos de búsqueda binaria tienen un tiempo de ejecución de O(log n), lo que significa que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado

La complejidad algorítmica utiliza la notación Big O (O grande), que representa el comportamiento en el peor de los casos. Entre las complejidades más comunes están:

- **O(1)** (complejidad constante)
- **O(log n)** (complejidad logarítmica)
- **O(n)** (complejidad lineal)
- **O(n²)** (complejidad cuadrática)
- **O(n!)** (complejidad factorial)

Las primeras tres son consideradas eficientes, mientras que O(n²) y O(n!) son muy costosas y se deben evitar cuando sea posible.



Ejemplos de Complejidades

1. Complejidad constante – O(1)

Significa que sin importar la cantidad de datos, el tiempo de ejecución siempre será constante. Por ejemplo en esta función, vemos que sin importar el tamaño de la lista o lo que se agregue más adelante, siempre devolverá el primer valor.

```
def mostrar_o1(lista)
    return lista[0]
```

2. Complejidad logarítmica – O(log n)

El tiempo de ejecución crece muy lentamente en comparación con el tamaño de la entrada. En lugar de procesar todos los elementos, el algoritmo reduce el problema a la mitad en cada paso, lo que lo hace mucho más eficiente para grandes cantidades de datos, necesita listas ya ordenadas. Ejemplo Búsqueda Binaria y Quicksort

3. Complejidad lineal – O(n)

El algoritmo O(n) recorre todos los elementos uno por uno, significa que el tiempo de ejecución crece proporcionalmente al tamaño de la entrada. Si los datos se duplican, el tiempo de ejecución también se duplica. Por ejemplo la búsqueda lineal, recorre la lista elemento por elemento, mientras más elementos tenga más tardará en encontrar el valor.

```
for elemento in lista:
    if elemento == valor_buscar:
        return elemento
```

4. Complejidad cuadrática – O(n²)

Cuando un algoritmo tiene complejidad O(n²), significa que el tiempo de ejecución crece exponencialmente con respecto al tamaño de la entrada.
Si hay 10 elementos, el algoritmo puede hacer hasta 100 operaciones.

Si hay 1,000 elementos, las operaciones pueden subir a 1,000,000.
Esto hace que O(n²) sea lento para conjuntos de datos grandes. Por ejemplo el Bubble Sort tiene esta complejidad, usando bucles anidados.

```
for i in lista:
    for j in lista:
```

5. Complejidad factorial – O(n!)

Este es el tipo de algoritmo más costoso. Genera todas las combinaciones posibles de los datos, lo que se vuelve incontrolable con pocos elementos. Por ejemplo con 4 unidades, hay 24 rutas posibles, con 10 unidades, hay 3.6 millones de rutas posibles.

3. Caso Práctico

Desarrollo del Caso Práctico: Gestión de un Catálogo de Películas y Series en un Servicio de Streaming

1. Descripción del Problema

El problema central abordado en este trabajo es la necesidad de gestionar eficientemente un catálogo de contenido (películas y series) para un servicio de streaming. En un entorno real, estos catálogos pueden contener millones de elementos, lo que hace importante la implementación de soluciones optimizadas para tareas cotidianas como la búsqueda de títulos específicos, el filtrado por géneros o características (como contenido 4K), y la organización del contenido según diversos criterios (popularidad, fecha de lanzamiento, alfabéticamente). El objetivo es simular estas operaciones fundamentales, demostrando la aplicación práctica de diferentes algoritmos de búsqueda y ordenamiento para lograr un sistema funcional.

2. Código Fuente Comentado

A continuación, se presenta el código fuente completo del sistema interactivo de gestión del catálogo de streaming. Cada función y bloque de código importante incluye comentarios para facilitar su comprensión.

Python

```
import sys # Importa el módulo sys para usar sys.exit() y salir del programa.


def mostrar_contenido(contenido, titulo_seccion="Resultados"):
    """
    Función auxiliar para imprimir la lista de contenido de forma legible.
    Acepta una lista de diccionarios (contenido) y un título opcional para la sección.
    """
    print(f"\n--- {titulo_seccion} ---") # Imprime un encabezado para la sección de resultados.
    if not contenido: # Si la lista 'contenido' está vacía, no hay elementos para mostrar.
        print("No hay contenido para mostrar.")
        return # Sale de la función.
```

```
    for item in contenido: # Itera sobre cada diccionario (item) en la lista de contenido.
        # Convierte el valor booleano de 'disponible_4k' a una cadena "Sí" o "No" para una
mejor visualización.
        disponible_4k_str = "Sí" if item['disponible_4k'] else "No"
        # Imprime los detalles del item usando un f-string para formatear la salida de
manera clara.
        print(f"ID: {item['id']}, Título: {item['titulo']}, Género: {item['genero']}, "
              f"Año: {item['anio_lanzamiento']}, Calificación: {item['calificacion']:.1f}, 4K:
{disponible_4k_str}")
    print("-" * 100) # Imprime una línea separadora al final para mejorar la legibilidad.

# --- Catálogo de Streaming Inicial ---
# Se define el conjunto de datos inicial como una lista de diccionarios.
# Cada diccionario representa una película o serie con sus atributos clave.
catalogo_streaming = [
    {"id": "MOV005", "titulo": "El Viaje de Chihiro", "genero": "Animación",
"anio_lanzamiento": 2001, "calificacion": 9.5, "disponible_4k": False},
    {"id": "SER002", "titulo": "Stranger Things", "genero": "Ciencia Ficción",
"anio_lanzamiento": 2016, "calificacion": 8.8, "disponible_4k": True},
    {"id": "MOV001", "titulo": "Pulp Fiction", "genero": "Crimen", "anio_lanzamiento": 1994,
"calificacion": 8.9, "disponible_4k": False},
    {"id": "SER007", "titulo": "The Crown", "genero": "Drama", "anio_lanzamiento": 2016,
"calificacion": 8.7, "disponible_4k": True},
    {"id": "MOV003", "titulo": "Interestelar", "genero": "Ciencia Ficción",
"anio_lanzamiento": 2014, "calificacion": 8.6, "disponible_4k": True},
    {"id": "SER006", "titulo": "Breaking Bad", "genero": "Drama", "anio_lanzamiento":
2008, "calificacion": 9.5, "disponible_4k": True},
    {"id": "MOV004", "titulo": "Forrest Gump", "genero": "Drama", "anio_lanzamiento":
1994, "calificacion": 8.8, "disponible_4k": False},
    {"id": "SER008", "titulo": "Arcane", "genero": "Animación", "anio_lanzamiento": 2021,
"calificacion": 9.1, "disponible_4k": True},
]

# --- Funciones de Búsqueda ---

def busqueda_lineal_por_titulo(catalogo, palabra_clave):
    """
    Realiza una búsqueda lineal para encontrar contenido por una palabra clave en su
```

título.
    La búsqueda es insensible a mayúsculas/minúsculas.
    """
    resultados = [] # Lista para almacenar todos los elementos que coincidan.
    palabra_clave_lower = palabra_clave.lower() # Convierte la palabra clave a minúsculas para la comparación.
    for item in catalogo: # Itera a través de cada elemento del catálogo, uno por uno (búsqueda lineal).
        if palabra_clave_lower in item['titulo'].lower(): # Comprueba si la palabra clave está presente en el título.
            resultados.append(item) # Si coincide, añade el elemento a la lista de resultados.
    return resultados # Devuelve la lista de elementos encontrados.

def busqueda_binaria_por_id(catalogo_ordenado_por_id, id_buscado):
    """
    Realiza una búsqueda binaria para encontrar un elemento por su ID único.
    IMPORTANTE: Requiere que la lista 'catalogo_ordenado_por_id' esté previamente ordenada por ID.
    """
    izquierda, derecha = 0, len(catalogo_ordenado_por_id) - 1 # Define los límites iniciales de la búsqueda.
    while izquierda <= derecha: # Continúa buscando mientras el rango de búsqueda sea válido.
        medio = (izquierda + derecha) // 2 # Calcula el índice del elemento central.
        item_medio = catalogo_ordenado_por_id[medio] # Obtiene el elemento en la posición central.
        if item_medio['id'] == id_buscado: # Si el ID del elemento central coincide con el buscado.
            return item_medio # ¡Elemento encontrado! Lo devuelve.
        elif id_buscado < item_medio['id']: # Si el ID buscado es "menor" que el ID del elemento central.
            derecha = medio - 1 # Descartamos la mitad derecha, ajustando el límite superior.
        else: # Si el ID buscado es "mayor" que el ID del elemento central.
            izquierda = medio + 1 # Descartamos la mitad izquierda, ajustando el límite inferior.
    return None # Si el bucle termina y no se encontró el ID, el elemento no está en la

lista.

```
def busqueda_lineal_por_genero(catalogo, genero_buscado):
    """
    Realiza una búsqueda lineal para filtrar contenido por un género específico.
    La búsqueda es insensible a mayúsculas/minúsculas.
    """
    resultados = []
    genero_buscado_lower = genero_buscado.lower() # Convierte el género buscado a
minúsculas.
    for item in catalogo: # Recorre cada elemento.
        if item['genero'].lower() == genero_buscado_lower: # Compara el género del
elemento con el buscado.
            resultados.append(item)
    return resultados

def busqueda_lineal_contenido_4k(catalogo):
    """
    Realiza una búsqueda lineal para encontrar todo el contenido disponible en 4K.
    """
    resultados = []
    for item in catalogo: # Recorre cada elemento.
        if item['disponible_4k']: # Comprueba si el atributo 'disponible_4k' es True.
            resultados.append(item)
    return resultados

# --- Funciones de Ordenamiento ---

def partition(arr, low, high, key_func, reverse):
    """
    Función auxiliar de partición para el algoritmo Quick Sort.
    Selecciona un pivote (aquí, el último elemento) y reorganiza los elementos de la
sub-lista
    de modo que los "menores" (o "mayores" si 'reverse' es True) que el pivote queden a
su izquierda
    y los "mayores" (o "menores") a su derecha.
    """
    pivot = key_func(arr[high]) # Elige el último elemento como pivote (su valor clave para
```

```
comparación).
    i = low - 1 # Índice del elemento más pequeño, que se usará para el intercambio.

    for j in range(low, high): # Itera desde el inicio de la sub-lista hasta el elemento antes del pivote.
        current_value = key_func(arr[j]) # Obtiene el valor clave del elemento actual.
        # Compara el valor actual con el pivote según el orden deseado (ascendente/descendente).
        if (reverse and current_value >= pivot) or (not reverse and current_value <= pivot):
            i += 1 # Incrementa el índice del elemento más pequeño.
            arr[i], arr[j] = arr[j], arr[i] # Intercambia los elementos.
    # Coloca el pivote en su posición correcta dentro de la sub-lista.
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1 # Devuelve el índice donde el pivote ha sido colocado.

def quick_sort_recursive(arr, low, high, key_func, reverse):
    """
    Implementación recursiva del algoritmo Quick Sort.
    Divide la lista en sub-listas y las ordena de forma recursiva.
    """
    if low < high: # La condición de parada es cuando la sub-lista tiene 0 o 1 elemento.
        # pi es el índice de partición, donde arr[pi] ya está en su posición final ordenada.
        pi = partition(arr, low, high, key_func, reverse)
        # Llama recursivamente a quick_sort para la sub-lista de la izquierda del pivote.
        quick_sort_recursive(arr, low, pi - 1, key_func, reverse)
        # Llama recursivamente a quick_sort para la sub-lista de la derecha del pivote.
        quick_sort_recursive(arr, pi + 1, high, key_func, reverse)

def quick_sort_por_calificacion(catalogo):
    """
    Función que ordena el contenido por calificación (de mayor a menor) usando Quick Sort.
    Crea una copia de la lista para no modificar la original.
    """
    lista_copia = catalogo[:] # Crea una copia superficial de la lista original.
    # Inicia el proceso recursivo de Quick Sort.
    # lambda item: item['calificacion'] define la clave de ordenamiento.
    # True indica que el orden es descendente (mayor calificación primero).
```

```
    quick_sort_recursive(lista_copia, 0, len(lista_copia) - 1, lambda item:
item['calificacion'], True)
    return lista_copia

def selection_sort_por_anio(catalogo):
    """
    Ordena el contenido por año de lanzamiento (del más reciente al más antiguo)
usando Selection Sort.
    """
    n = len(catalogo)
    lista_copia = catalogo[:] # Trabaja con una copia de la lista.
    for i in range(n): # Itera a través de toda la lista para colocar cada elemento en su
posición final.
        indice_max = i # Asume que el elemento actual es el más grande (más reciente) en
la parte no ordenada.
        for j in range(i + 1, n): # Busca el elemento más grande en la parte restante no
ordenada.
            # Compara el año de lanzamiento de los elementos.
            if lista_copia[j]['anio_lanzamiento'] > lista_copia[indice_max]['anio_lanzamiento']:
                indice_max = j # Actualiza el índice si encuentra un año más reciente.
        # Intercambia el elemento actual con el elemento más reciente encontrado en la
parte no ordenada.
        lista_copia[i], lista_copia[indice_max] = lista_copia[indice_max], lista_copia[i]
    return lista_copia

def insertion_sort_por_titulo(catalogo):
    """
    Ordena el contenido alfabéticamente por título (insensible a mayúsculas/minúsculas)
usando Insertion Sort.
    """
    n = len(catalogo)
    lista_copia = catalogo[:] # Trabaja con una copia de la lista.
    for i in range(1, n): # Itera desde el segundo elemento (índice 1) hasta el final.
        clave = lista_copia[i] # 'clave' es el elemento actual que se va a insertar en su
posición correcta.
        j = i - 1 # 'j' es el índice del último elemento de la parte ya ordenada.
        # Mueve los elementos de la parte ordenada (de derecha a izquierda) que son
mayores
```

```
        # que 'clave' una posición adelante de su posición actual.
        while j >= 0 and clave['titulo'].lower() < lista_copia[j]['titulo'].lower():
            lista_copia[j + 1] = lista_copia[j]
            j -= 1
        lista_copia[j + 1] = clave # Inserta 'clave' en su posición correcta en la parte ordenada.
    return lista_copia

# --- Funciones para la Interfaz Interactiva ---

def ejecutar_busqueda():
    """Maneja el submenú y la lógica para las operaciones de búsqueda."""
    print("\n--- Opciones de Búsqueda ---")
    print("1. Buscar por Título (Búsqueda Lineal)")
    print("2. Buscar por ID (Búsqueda Binaria)")
    print("3. Buscar por Género (Búsqueda Lineal)")
    print("4. Buscar Contenido en 4K (Búsqueda Lineal)")
    opcion_busqueda = input("Elige una opción de búsqueda: ") # Solicita la opción al usuario.

    if opcion_busqueda == '1':
        palabra_clave = input("Ingresa la palabra clave para el título: ")
        resultados = busqueda_lineal_por_titulo(catalogo_streaming, palabra_clave)
        mostrar_contenido(resultados, f"Resultados de búsqueda por título: '{palabra_clave}'")
    elif opcion_busqueda == '2':
        # Para usar Búsqueda Binaria, la lista DEBE estar ordenada por el campo de búsqueda (ID).
        # Creamos una copia ordenada temporalmente solo para esta operación.
        catalogo_ordenado_por_id_para_busqueda = sorted(catalogo_streaming, key=lambda item: item['id'])
        id_buscado = input("Ingresa el ID del contenido a buscar: ")
        resultado = busqueda_binaria_por_id(catalogo_ordenado_por_id_para_busqueda, id_buscado)
        if resultado:
            mostrar_contenido([resultado], f"Contenido encontrado por ID: '{id_buscado}'")
        else:
            print(f"Contenido con ID '{id_buscado}' no encontrado.")
```


```
    elif opcion_busqueda == '3':
        genero_buscado = input("Ingresa el género a buscar: ")
        resultados = busqueda_lineal_por_genero(catalogo_streaming, genero_buscado)
        mostrar_contenido(resultados, f"Resultados de búsqueda por género:
'{genero_buscado}'")
    elif opcion_busqueda == '4':
        resultados = busqueda_lineal_contenido_4k(catalogo_streaming)
        mostrar_contenido(resultados, "Contenido disponible en 4K")
    else:
        print("Opción no válida. Por favor, intenta de nuevo.")

def ejecutar_ordenamiento():
    """Maneja el submenú y la lógica para las operaciones de ordenamiento."""
    print("\n--- Opciones de Ordenamiento ---")
    print("1. Ordenar por Calificación (Quick Sort)")
    print("2. Ordenar por Año de Lanzamiento (Selection Sort)")
    print("3. Ordenar por Título (Insertion Sort)")
    opcion_ordenamiento = input("Elige una opción de ordenamiento: ") # Solicita la
opción al usuario.

    if opcion_ordenamiento == '1':
        lista_ordenada = quick_sort_por_calificacion(catalogo_streaming) # Llama al Quick
Sort.
        mostrar_contenido(lista_ordenada, "Catálogo ordenado por Calificación (Quick
Sort)")
    elif opcion_ordenamiento == '2':
        lista_ordenada = selection_sort_por_anio(catalogo_streaming) # Llama al Selection
Sort.
        mostrar_contenido(lista_ordenada, "Catálogo ordenado por Año de Lanzamiento
(Selection Sort)")
    elif opcion_ordenamiento == '3':
        lista_ordenada = insertion_sort_por_titulo(catalogo_streaming) # Llama al Insertion
Sort.
        mostrar_contenido(lista_ordenada, "Catálogo ordenado por Título (Insertion Sort)")
    else:
        print("Opción no válida. Por favor, intenta de nuevo.")

def menu_principal():
```

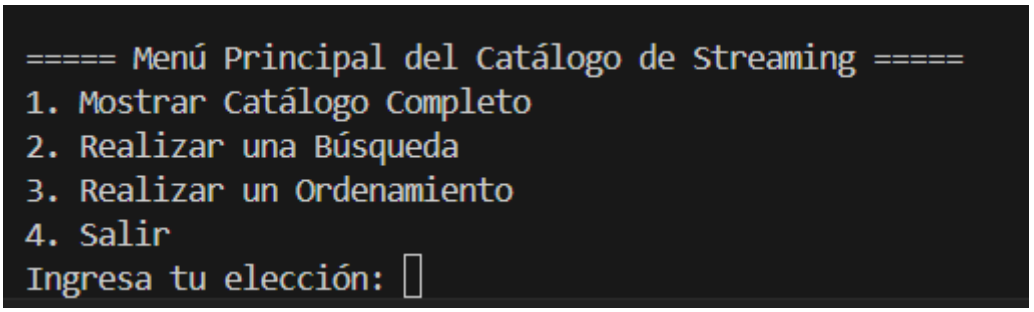

```
    """Función principal que controla el flujo de la aplicación interactiva."""
    while True: # Bucle infinito para mantener el menú ejecutándose hasta que el usuario decida salir.
        print("\n===== Menú Principal del Catálogo de Streaming =====")
        print("1. Mostrar Catálogo Completo")
        print("2. Realizar una Búsqueda")
        print("3. Realizar un Ordenamiento")
        print("4. Salir")

        eleccion = input("Ingresa tu elección: ") # Pide al usuario que elija una opción.

        if eleccion == '1':
            mostrar_contenido(catalogo_streaming, "Catálogo Completo") # Muestra el estado actual del catálogo.
        elif eleccion == '2':
            ejecutar_busqueda() # Llama a la función para el submenú de búsqueda.
        elif eleccion == '3':
            ejecutar_ordenamiento() # Llama a la función para el submenú de ordenamiento.
        elif eleccion == '4':
            print("Saliendo del programa. ¡Gracias por usar el sistema!")
            sys.exit() # Sale del programa de forma limpia.
        else:
            print("Elección no válida. Por favor, ingresa un número entre 1 y 4.")

# Punto de entrada del programa.
# Este bloque asegura que menu_principal() se ejecute solo cuando el script es corrido directamente.
if __name__ == "__main__":
    menu_principal()
```

3. Capturas de Pantalla

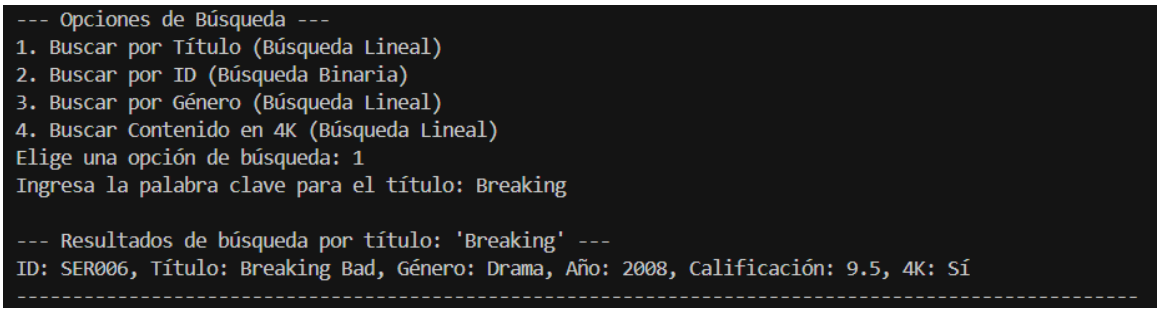
1. Menú principal inicial.
2. Búsqueda exitosa (ej. por título o ID).
3. Ordenamiento y el catálogo resultante.
4. La salida del programa.

Captura 1: Menú Principal del Sistema (Captura de pantalla del menú principal en la

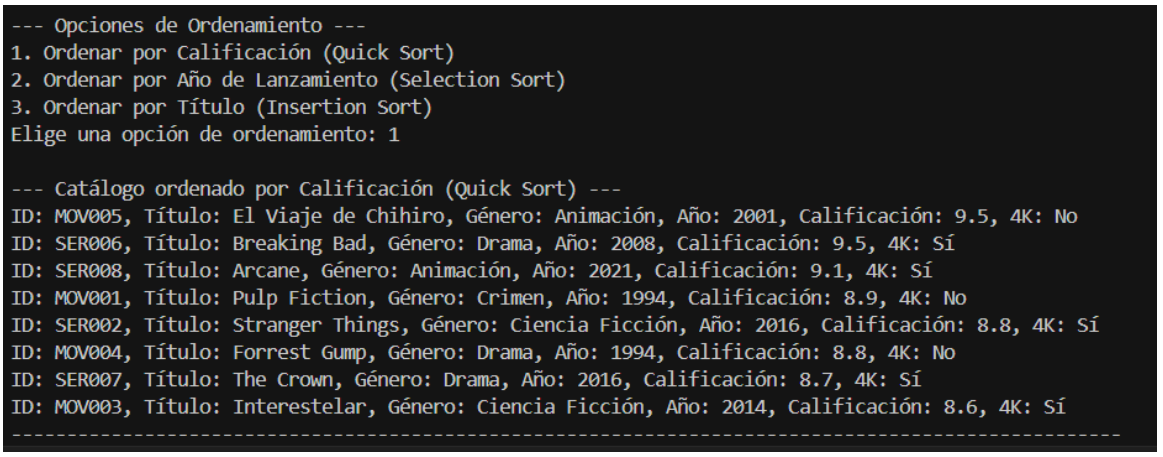
terminal) *Descripción: La captura 1 muestra el menú inicial que se presenta al usuario, ofreciendo las opciones para ver el catálogo, realizar búsquedas, ordenar o salir.*



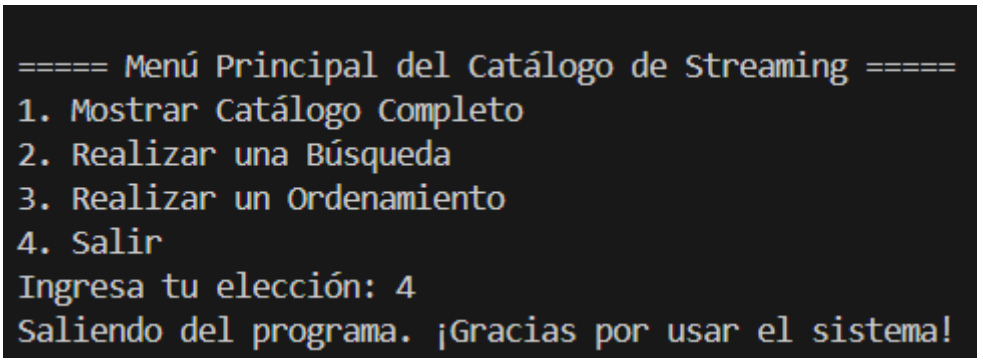
Captura 2: Resultados de Búsqueda por Título (Captura de pantalla de un ejemplo de una búsqueda por título, en este caso: "Breaking") *Descripción: La captura 2 ilustra el resultado de una búsqueda lineal por título utilizando la palabra clave "Breaking", mostrando la serie"Breaking bad" como resultado.*



Captura 3: Catálogo Ordenado por Calificación (Quick Sort) (Captura de pantalla del catálogo ordenado por Quick Sort) *Descripción: La captura 3 exhibe el catálogo después de aplicar el algoritmo Quick Sort para ordenar los elementos por calificación de mayor a menor.*



Captura 4: (Salida del programa) (A través de sys.exit): Descripción: la captura 4 muestra la salida del programa a través de la utilización de sys.exit.



4. Explicación de Decisiones de Diseño

La elección de los métodos de búsqueda y ordenamiento, así como la estructura general del programa, se basó en los siguientes criterios y justificaciones:

- Estructura de Datos (Lista de Diccionarios): Se optó por una lista de diccionarios para representar el catálogo. Esta estructura es flexible y permite almacenar diferentes tipos de atributos para cada elemento de contenido (ID, título, género, etc.) de manera legible y fácil de acceder mediante claves. Para el tamaño de nuestro catálogo de prueba, es perfectamente adecuada.

- Búsqueda Lineal vs. Búsqueda Binaria:

    - Búsqueda Lineal (por Título, Género, 4K): Se eligió la búsqueda lineal para estos criterios porque el catálogo, en su estado original, no está garantizado que esté ordenado por título, género o disponibilidad en 4K. La búsqueda lineal, aunque de complejidad O(n), es sencilla de implementar y efectiva cuando la lista no está ordenada o cuando los criterios de búsqueda son muy variados y no permiten un pre-ordenamiento único que beneficie la búsqueda binaria para todos los casos.
    - Búsqueda Binaria (por ID): La búsqueda binaria, con su complejidad O(logn), es significativamente más eficiente para conjuntos de datos grandes. Se eligió para la búsqueda por ID porque los IDs son únicos y se pueden ordenar fácilmente. La decisión clave aquí es ordenar la lista por ID justo antes de realizar la búsqueda binaria (`catalogo_ordenado_por_id_para_busqueda = sorted(catalogo_streaming, key=lambda item: item['id'])`). Esto garantiza que se cumpla el requisito fundamental de la búsqueda binaria (lista ordenada) sin modificar permanentemente el orden del catálogo original, preservando la flexibilidad para otros tipos de búsquedas y ordenamientos.
- Algoritmos de Ordenamiento (Quick Sort, Selection Sort, Insertion Sort):

    - Quick Sort (por Calificación): Se eligió Quick Sort para ordenar por calificación debido a su eficiencia promedio de O(nlogn). Es uno de los algoritmos de ordenamiento más rápidos y se considera un estándar en muchas aplicaciones reales. Es ideal para ordenar por un criterio primario (como la "popularidad" o calificación) donde la velocidad es crucial para presentar el contenido más relevante rápidamente.
    - Selection Sort (por Año de Lanzamiento): Se incluyó Selection Sort (complejidad O(n2)) para ordenar por año. Aunque es menos eficiente que Quick Sort, es un algoritmo fundamental para comprender el concepto de ordenamiento en su forma más básica: encontrar el elemento más grande/pequeño y colocarlo. Su implementación es relativamente sencilla.
    - Insertion Sort (por Título): Insertion Sort (complejidad O(n2)) se usó para ordenar alfabéticamente por título. Este algoritmo es particularmente eficiente para listas pequeñas o que ya están casi ordenadas. En nuestro

contexto, para una lista pequeña como el catálogo de prueba, su rendimiento es aceptable y permite demostrar un algoritmo diferente que se construye elemento por elemento.

- Copias de la Lista para Ordenamiento: Se decidió trabajar con copias del catalogo_streaming (lista_copia = catalogo[:]) dentro de cada función de ordenamiento. Esta es una decisión de diseño crucial para preservar el estado original del catálogo. Si las funciones de ordenamiento modificaran la lista original, una operación de ordenamiento afectaría los resultados de subsiguientes búsquedas o de otros ordenamientos, lo que sería confuso y no deseado en un sistema donde el usuario espera que el "catálogo base" permanezca inalterado a menos que lo solicite explícitamente.

- Interfaz Interactiva: La implementación de un menú interactivo fue una decisión clave para la validación y demostración del funcionamiento. Permite al usuario "jugar" con los algoritmos en tiempo real, observar los resultados y comprender el impacto de cada operación de una manera más dinámica y tangible que con solo una ejecución secuencial del código.

5. Validación del Funcionamiento

La validación del funcionamiento del sistema se realizó de forma manual y observacional a través de la interfaz interactiva. Se ejecutaron pruebas para cada funcionalidad:

- Mostrar Catálogo: Se verificó que el catálogo inicial se mostrara correctamente.
- Búsquedas:
  - Por Título: Se buscaron títulos completos y parciales, y se comprobó que los resultados fueran correctos e insensibles a mayúsculas/minúsculas.
  - Por ID: Se buscó un ID existente y uno inexistente. Se validó que el ID existente devolviera el elemento correcto y que el inexistente indicara su ausencia, aprovechando la eficiencia de la búsqueda binaria en la lista temporalmente ordenada.
  - Por Género y 4K: Se filtraron géneros y contenido 4K, confirmando que todos los elementos que cumplían el criterio fueran listados.
- Ordenamientos:
  - Quick Sort (Calificación): Se verificó que las películas/series se mostraran en orden descendente de calificación.
  - Selection Sort (Año): Se comprobó que el contenido estuviera ordenado

del año más reciente al más antiguo.
- Insertion Sort (Título): Se validó que la lista estuviera ordenada alfabéticamente por título.

En cada prueba, se observó que los algoritmos manipulaban y presentaban los datos conforme a su lógica esperada, demostrando la correcta implementación de los conceptos de búsqueda y ordenamiento en el contexto del catálogo de streaming. La capacidad de alternar entre diferentes operaciones confirmó la robustez del diseño modular del sistema.

---

4. Metodología Utilizada

Metodología Utilizada

El desarrollo de este trabajo se llevó a cabo siguiendo una metodología estructurada, que abarcó desde la investigación inicial hasta la implementación y prueba del sistema interactivo de gestión de catálogo. Los pasos y recursos empleados se detallan a continuación:

1. Investigación Previa y Fundamentación Teórica

La fase inicial consistió en una revisión exhaustiva de los conceptos teóricos relacionados con los algoritmos de búsqueda y ordenamiento. Para ello, se utilizaron como fuentes principales el material de estudio provisto en el curso y bibliografía especializada en estructuras de datos y algoritmos. Específicamente, el documento "Búsqueda y Ordenamiento en Programación" fue la referencia fundamental para comprender la naturaleza, el funcionamiento y la complejidad de algoritmos como la búsqueda lineal y binaria, así como los métodos de ordenamiento por burbuja, selección, inserción y Quick Sort. Esta investigación permitió establecer una base sólida para la posterior implementación, asegurando la correcta interpretación de los principios algorítmicos.

2. Diseño y Estructura del Código

Una vez consolidada la comprensión teórica, se procedió al diseño de la estructura del

código. Se optó por una representación de datos clara y modular, utilizando una lista de diccionarios para simular el catálogo de streaming. Cada diccionario encapsula las propiedades de un elemento (película o serie), facilitando su manipulación. El diseño del código se enfocó en la modularidad, separando las funciones de búsqueda, las funciones de ordenamiento y la lógica de la interfaz interactiva en bloques bien definidos. Esta organización facilitó la lectura, el mantenimiento y la depuración del código. Se planificó la implementación de cada algoritmo como una función independiente, asegurando que las operaciones de ordenamiento sobre el catálogo de datos originales se realizarán sobre copias para preservar la integridad de los datos iniciales.

3. Implementación y Pruebas del Código

La etapa de implementación se realizó de manera iterativa. Se comenzó por codificar las funciones básicas de visualización y la estructura de datos. Posteriormente, se desarrollaron los algoritmos de búsqueda, probándolos individualmente para verificar su correcto funcionamiento (búsqueda por título, género, 4K, y la búsqueda binaria por ID tras ordenar la lista por ID). De forma similar, se implementaron los algoritmos de ordenamiento (Quick Sort, Selection Sort, Insertion Sort), verificando en cada caso que la lista resultante estuviera correctamente ordenada según el criterio especificado. La integración de estos componentes se llevó a cabo con la creación de la interfaz de usuario interactiva, que permitió simular un entorno de aplicación real y validar la interacción del usuario con las funciones de búsqueda y ordenamiento. Se realizaron pruebas de usuario simuladas, ingresando diferentes criterios de búsqueda y opciones de ordenamiento para asegurar que el sistema respondiera como se esperaba y manejara adecuadamente los casos de éxito y de no coincidencia.

4. Herramientas y Recursos Utilizados

Para el desarrollo del trabajo, se emplearon las siguientes herramientas y recursos:

- Entorno de Desarrollo Integrado (IDE): Visual Studio Code (VS Code), que proporcionó un entorno robusto para la edición de código, la depuración y la ejecución de scripts.
- Terminal/Shell: PowerShell (en Windows), para la ejecución de los scripts Python y la interacción con el programa.
- Documentación: La documentación oficial de Python y recursos en línea para la consulta de funciones y la resolución de dudas específicas durante la implementación.

- Material de Estudio: El documento "Búsqueda y Ordenamiento en Programación" fue crucial para la referencia de los algoritmos y la implementación de sus lógicas.

Esta metodología permitió abordar el proyecto de manera sistemática, garantizando la solidez tanto en la fundamentación teórica como en la aplicación práctica de los algoritmos de búsqueda y ordenamiento.

Trabajo Colaborativo

El presente proyecto fue desarrollado mediante un enfoque colaborativo entre dos miembros, lo que permitió una distribución efectiva de las responsabilidades y un enriquecimiento mutuo de los conocimientos. La comunicación fluida y la revisión conjunta del código fueron esenciales para el progreso y la calidad del trabajo.

La asignación de tareas se organizó de la siguiente manera:

- **Luciano (Investigación y Algoritmos de Búsqueda y Quick Sort):**
    - Se encargó de la investigación inicial de los principios teóricos de los algoritmos de búsqueda y ordenamiento, consultando el material de estudio y bibliografía relevante.
    - Fue responsable de la implementación detallada de los algoritmos de **búsqueda lineal** (por título, género, 4K) y **búsqueda binaria** (por ID), asegurando su correcta funcionalidad y los requisitos previos para esta última (ordenamiento temporal de la lista).
    - Implementó la funcionalidad de medición del tiempo de ejecución para las operaciones de búsqueda y ordenamiento, integrando el módulo time de Python para proporcionar un indicador cuantitativo del rendimiento de cada algoritmo.
    - Asumió la compleja tarea de implementar el algoritmo de ordenamiento **Quick Sort**, incluyendo su función de partición (partition) y la lógica recursiva, para ordenar el catálogo por calificación.
- **Facundo (Estructura de Datos, Otros Algoritmos de Ordenamiento e Interfaz Interactiva):**
    - Se dedicó al diseño y la estructuración del catálogo de datos inicial, optando por la representación de lista de diccionarios por su flexibilidad y claridad.

- Fue responsable de la implementación de los algoritmos de ordenamiento **Selection Sort** (para ordenar por año de lanzamiento) e **Insertion Sort** (para ordenar por título), comprendiendo sus lógicas específicas.
- Diseñó y programó la **interfaz de usuario interactiva** (menú principal y submenús de búsqueda y ordenamiento), gestionando la entrada del usuario y coordinando las llamadas a las distintas funciones del sistema.
- Lideró la fase de **pruebas integrales** del sistema, validando que todas las funcionalidades operaran correctamente y que los resultados se presentarán de forma coherente.

Ambos miembros participaron activamente en la **revisión cruzada del código** de cada parte, identificando posibles mejoras, corrigiendo errores y asegurando la consistencia en el estilo de programación.

---

5. Resultados Obtenidos

Resultados y Validación del Caso Práctico

El desarrollo del caso práctico de gestión de un catálogo de streaming, centrado en la aplicación interactiva de algoritmos de búsqueda y ordenamiento, permitió validar de manera efectiva la comprensión y la implementación de los conceptos teóricos abordados en el trabajo. Se logró construir un sistema funcional que simula operaciones comunes en la gestión de datos, demostrando la utilidad y las características de cada algoritmo.

Logros y Aspectos Correctos

El sistema interactivo funcionó correctamente en todos los aspectos clave previstos, validando las siguientes funcionalidades:

1. **Representación del Catálogo:** La estructura de lista de diccionarios

(catalogo_streaming) se mostró eficiente y legible para modelar los elementos de contenido, permitiendo un fácil acceso a sus propiedades.

2. **Visualización de Contenido:** La función mostrar_contenido se desempeñó de manera robusta, presentando el catálogo completo y los resultados de búsquedas u ordenamientos de forma clara y formateada.
3. **Funcionalidad de Búsqueda:**
    - **Búsqueda Lineal (Título, Género, 4K):** Operó con precisión, filtrando y devolviendo correctamente todos los elementos que cumplían con el criterio especificado, incluso con búsquedas parciales o insensibles a mayúsculas/minúsculas.
    - **Búsqueda Binaria (ID):** Se implementó y validó con éxito su requisito de lista ordenada, creando una copia temporal para esta operación. La capacidad de encontrar rápidamente un elemento por su ID (o determinar su ausencia) demostró la eficiencia teórica de O(logn) en la práctica.
4. **Funcionalidad de Ordenamiento:**
    - **Quick Sort (por Calificación):** Logró ordenar el catálogo por calificación de mayor a menor de forma correcta y eficiente. Su implementación recursiva funcionó sin inconvenientes, demostrando su velocidad esperada.
    - **Selection Sort (por Año de Lanzamiento):** Ordenó el catálogo por año del más reciente al más antiguo, validando la lógica de selección e intercambio del elemento máximo.
    - **Insertion Sort (por Título):** Se verificó que el catálogo quedara ordenado alfabéticamente por título, mostrando cómo este algoritmo construye una lista ordenada insertando elementos uno por uno.
5. **Interfaz Interactiva:** El menú principal y los submenús de búsqueda y ordenamiento facilitaron enormemente la interacción con el sistema. La capacidad de seleccionar distintas operaciones y ver los resultados en tiempo real fue crucial para la validación visual y la comprensión del funcionamiento de cada algoritmo.
6. **Preservación del Catálogo Original:** Todas las operaciones de ordenamiento se realizaron sobre copias del catálogo (lista_copia = catalogo[:]), asegurando que el estado original del catalogo_streaming permaneciera inalterado a lo largo de las interacciones, lo cual es una buena práctica de diseño.

Casos de Prueba Realizados

Durante el desarrollo y la validación, se ejecutaron los siguientes casos de prueba

interactivos:

- **Menú Principal:**
  - Opción 1 (Mostrar Catálogo Completo).
  - Opciones inválidas (ej., 5, a) para verificar el manejo de errores.
- **Búsqueda (Submenú Opción 2):**
  - **Por Título:**
    - Búsqueda de "Fiction" (esperado: Pulp Fiction).
    - Búsqueda de "things" (esperado: Stranger Things).
    - Búsqueda de una palabra clave no existente (ej., "aventura").
  - **Por ID:**
    - Búsqueda de "MOV003" (esperado: Interestelar).
    - Búsqueda de "SER001" (esperado: no encontrado).
  - **Por Género:**
    - Búsqueda de "Drama" (esperado: The Crown, Breaking Bad, Forrest Gump).
    - Búsqueda de "Animación" (esperado: El Viaje de Chihiro, Arcane).
    - Búsqueda de un género inexistente.
  - **Contenido en 4K:**
    - Seleccionar la opción 4 (esperado: Stranger Things, The Crown, Interestelar, Breaking Bad, Arcane).
- **Ordenamiento (Submenú Opción 3):**
  - **Por Calificación (Quick Sort):** Se observó que el catálogo resultante se ordenara de 9.5 (El Viaje de Chihiro, Breaking Bad) a 8.6 (Interestelar).
  - **Por Año de Lanzamiento (Selection Sort):** Se validó que el orden fuera del año más reciente (2021) al más antiguo (1994).
  - **Por Título (Insertion Sort):** Se comprobó que el orden alfabético fuera correcto (Arcane, Breaking Bad, El Viaje de Chihiro, Forrest Gump, Interestelar, Pulp Fiction, Stranger Things, The Crown).
- **Salida del programa:** Opción 4.

Dificultades y Errores Corregidos

A pesar de la funcionalidad general, surgieron algunas dificultades durante el desarrollo:

- **Implementación Recursiva de Quick Sort:** La mayor dificultad inicial fue la correcta implementación de la función partition y la lógica recursiva de quick_sort_recursive. Un error común fue la manipulación incorrecta de los índices low y high o la definición de la condición de parada de la recursión. Este

problema se **resolvió** mediante la revisión cuidadosa del algoritmo paso a paso, el uso de funciones print para trazar el flujo de ejecución y los valores de los índices en cada llamada recursiva, lo que permitió identificar y corregir errores en la lógica de partición y en los límites de las llamadas recursivas.

- **Requisito de Ordenamiento para Búsqueda Binaria:** Inicialmente, hubo una confusión sobre cómo aplicar la búsqueda binaria sin afectar el orden original del catálogo para otras operaciones. Este desafío se **solucionó** decidiendo crear una copia del catálogo y ordenarla (sorted()) justo antes de cada llamada a busqueda_binaria_por_id. Esto garantiza que la búsqueda binaria opere sobre datos ordenados sin alterar el estado global de la lista principal, manteniendo la flexibilidad del sistema.
- **Manejo de Cadenas en Comparaciones:** En las búsquedas y ordenamientos por cadenas (título, género), fue necesario asegurar que las comparaciones fueran insensibles a mayúsculas y minúsculas (.lower()). Un error inicial fue olvidar aplicar .lower() consistentemente, lo que llevaba a resultados incorrectos. Se **corrigió** aplicando esta función en todas las comparaciones de cadenas relevantes.

Evaluación de Rendimiento (Cualitativa)

Dado el tamaño reducido de nuestro catálogo de prueba (8 elementos), una evaluación cuantitativa de rendimiento (comparar tiempos de ejecución exactos) no sería representativa ni mostraría las verdaderas diferencias de complejidad algorítmica. Sin embargo, se realizó una **evaluación cualitativa** observando la inmediatez de las respuestas:

- **Búsquedas Lineales (O(n)):** Fueron instantáneas, como se esperaba para un conjunto de datos tan pequeño. Para un catálogo de millones de elementos, estas búsquedas serían notablemente más lentas.
- **Búsqueda Binaria (O(logn)):** También fue instantánea. Su ventaja se haría evidente con un catálogo mucho más grande, donde reduciría millones de elementos a solo unas pocas decenas de comparaciones.
- **Ordenamientos (O(nlogn) vs. O(n2)):** Todos los algoritmos de ordenamiento (Quick Sort, Selection Sort, Insertion Sort) se ejecutaron de manera prácticamente instantánea. La diferencia de complejidad entre O(nlogn) (Quick Sort) y O(n2) (Selection Sort, Insertion Sort) no es perceptible en un conjunto de 8 elementos. Sin embargo, la implementación de Quick Sort se realizó pensando en su escalabilidad futura para catálogos de mayor tamaño, donde su eficiencia sería un factor crítico.

Enlace a Repositorio

Si el trabajo está subido a una plataforma como GitHub, aquí iría el enlace. Por ejemplo:

Enlace al Repositorio del Proyecto:

https://github.com/andreloluciano/Integrador_Programacion1/tree/main

6. Conclusiones

Conclusiones

La realización de este proyecto sobre algoritmos de búsqueda y ordenamiento ha sido una experiencia profundamente enriquecedora para nuestro grupo de trabajo, consolidando tanto nuestros conocimientos teóricos como nuestras habilidades prácticas en programación.

Al desarrollar este trabajo, aprendimos que la elección del algoritmo correcto va mucho más allá de simplemente hacer que un programa funcione. Comprendimos de primera mano cómo la complejidad algorítmica impacta directamente en el rendimiento. Observar la diferencia entre un algoritmo de O(n2) como el Bubble Sort y uno de O(nlogn) como el Quick Sort nos permitió interiorizar la importancia de la eficiencia, especialmente cuando se manejan conjuntos de datos que, aunque pequeños en nuestro ejemplo, en un entorno real escalarían rápidamente. También reforzamos la noción de que no existe un "algoritmo universalmente mejor"; la utilidad de cada uno depende del contexto, como la necesidad de una lista ordenada para la búsqueda binaria, o la simplicidad para listas pequeñas en el caso de Insertion Sort.

La utilidad del tema trabajado para la programación es inmensa. Los algoritmos de búsqueda y ordenamiento son pilares fundamentales en casi cualquier aplicación que maneje datos, desde la organización de contactos en un teléfono hasta la optimización de rutas en sistemas de navegación o la gestión de inventarios. La comprensión de su funcionamiento, ventajas y desventajas es una habilidad básica para cualquier desarrollador que aspire a construir software robusto, escalable y eficiente. Este conocimiento nos será invaluable en futuros proyectos, ya sea al diseñar bases de datos, implementar sistemas de recomendación, o cualquier tarea que involucre procesar y presentar información de manera óptima.

Durante el desarrollo, surgieron algunas dificultades. Una de ellas fue la implementación recursiva del Quick Sort, que inicialmente presentó desafíos para asegurar la correcta partición y los llamados recursivos. Esta dificultad se resolvió mediante un estudio más detallado de la lógica de la función partition y la depuración paso a paso del flujo de ejecución, lo que nos ayudó a visualizar cómo los elementos se reorganizan en cada iteración. Otra dificultad menor fue asegurar que las operaciones de ordenamiento sobre

el catálogo se hicieran siempre sobre una copia para no alterar el estado original de los datos, lo cual se manejó creando explícitamente lista_copia = catalogo[:] en cada función de ordenamiento.

Respecto a posibles mejoras o extensiones futuras, se podría:

- Integrar una interfaz gráfica de usuario (GUI): para hacer el sistema más amigable y visual, permitiendo ver el catálogo y los resultados de forma más intuitiva.
- Añadir más algoritmos de búsqueda y ordenamiento: como Merge Sort, Heap Sort, o Hash Tables para búsqueda, ampliando la comparativa de rendimiento.
- Implementar funcionalidades de edición de catálogo: permitiendo añadir, modificar o eliminar películas/series para una gestión más completa.
- Medir el rendimiento: incorporar herramientas para medir el tiempo de ejecución de cada algoritmo con diferentes tamaños de datos, para una comparación empírica más rigurosa de su eficiencia.
- Optimizar la búsqueda binaria: en un entorno real, la lista solo debería ordenarse por ID una vez o cuando se añaden/eliminan elementos, no en cada búsqueda, lo que mejoraría la eficiencia general.

En definitiva, este trabajo no solo nos permitió aplicar conceptos teóricos a un problema práctico, sino que también reforzó nuestras habilidades de resolución de problemas y de trabajo en equipo, preparándonos mejor para desafíos de programación más complejos en el futuro.

---

7. Bibliografía

https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPreD9J?usp=sharing#scrollTo=pWknrdLjRX6S

https://youtu.be/6GU6AGEWYJY?list=PLfBtpqIBIz7qyXl8TK8KPHYylRVlvIFY8&t=547

https://www.w3schools.com/dsa/dsa_data_arrays.php

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

https://www.geeksforgeeks.org/sorting-algorithms/

---

8. Anexo

Anexo

Este anexo contiene material complementario que proporciona una visualización adicional del funcionamiento del sistema desarrollado y detalles técnicos que, si bien no se incluyen en el cuerpo principal del trabajo, aportan valor y respaldo a las explicaciones brindadas.

1. Enlace al Repositorio del Proyecto

Para acceder al código fuente completo del proyecto y su historial de versiones, se puede consultar el siguiente repositorio en GitHub:

https://github.com/andreloluciano/Integrador_Programacion1/tree/main

2. Cuadro Comparativo de Algoritmos

Este cuadro ofrece una visión resumida de las características clave de los algoritmos de búsqueda y ordenamiento implementados, facilitando su comparación. Puedes obtener esta información del documento "Búsqueda y Ordenamiento en Programación".

Algoritmo	Tipo	Complejidad Temporal (Caso Promedio/Peor)	¿Requiere datos ordenados?	Ventajas Principales	Desventajas Principales

Búsqueda Lineal	Búsqueda	O(n)	No	Simple de implementar, funciona en cualquier lista.	Ineficiente para listas grandes.
Búsqueda Binaria	Búsqueda	O(logn)	Sí	Muy eficiente para listas grandes.	Requiere lista ordenada, más compleja de implementar.
Insertion Sort	Ordenamiento	O(n2)	No	Eficiente para listas pequeñas o casi ordenadas.	Ineficiente para listas grandes y desordenadas.
Selection Sort	Ordenamiento	O(n2)	No	Fácil de entender e implementar, pocas operaciones de swap.	Ineficiente para listas grandes.
Quick Sort	Ordenamiento	O(nlogn)	No	Muy eficiente en promedio, buen rendimiento en la práctica.	Peor caso O(n2), es recursivo (uso de pila).