

**Programa Educativo**

Ingeniería Informática.

**Facultad de Ingeniería Eléctrica y Electrónica.**

**Experiencia Educativa**

Diseño Y Análisis de Algoritmos

**Trabajo**

Proyecto final: Sistema de gestión de streamming.

**Estudiantes**

Ramos Garcia Julian.

Zarate Lozano Jehiel Antonio.

**Docente**  
Fleitas Toranzo Yadira.

**Veracruz, Veracruz. 02 de diciembre de 2024**

Contenido

[**Agrupación de películas por género** I](#_Toc184336109)

[**Top 10 películas más populares por calificación** VI](#_Toc184336110)

[**Películas ordenadas por calificación (1-5 estrellas), de mayor a menor.** IX](#_Toc184336111)

[**Cantidad de películas que se pueden ver para acumular un tiempo exacto.** XIV](#_Toc184336112)

[**Búsqueda por título o palabras clave** XVIII](#_Toc184336113)

[**Historial películas.** XXII](#_Toc184336114)

# **Agrupación de películas por género**

**Complejidad requerida: O(nlogn)**

Para esta funcionalidad del programa, decidimos usar el algoritmo **MergeSort**, ya que cumple con la complejidad buscada O(nlogn) que es eficiente para ordenar grandes cantidades de datos. Merge garantiza un rendimiento estable, independientemente del estado inicial del array, ya que siempre realiza el mismo número de comparaciones y fusiones.

**Descripción del algoritmo utilizado**

MergeSort es un algoritmo de **divide y vencerás** que funciona dividiendo el problema en subproblemas más pequeños, resolviéndolos de forma independiente y combinando sus soluciones para formar la solución global. En este caso, el array de películas se divide repetidamente hasta que los subarrays son de tamaño 1. Luego, estos subarrays se fusionan en orden utilizando el criterio de comparación por género.

**Justificación de la elección del algoritmo**

Elegimos MergeSort debido a las siguientes razones:

* **Eficiencia garantizada**: Su complejidad temporal es O(nlogn) en el mejor, peor y caso promedio, lo que lo hace predecible y adecuado para trabajar con grandes conjuntos de datos.
* **Estabilidad**: MergeSort es un algoritmo estable, lo que significa que mantiene el orden relativo de elementos iguales, una característica importante cuando se trabaja con estructuras como películas con múltiples atributos.
* **Adaptabilidad**: Aunque usa memoria adicional para los subarrays temporales, esto no representa un problema crítico para este programa, dado el tamaño esperado de los datos.

**Explicación del funcionamiento del algoritmo**

1. **División**:
   * El array principal se divide recursivamente en dos mitades hasta que cada subarray contiene un solo elemento.
   * El número de niveles de división es proporcional a log2(n) donde n es el tamaño del array inicial.
2. **Fusión**:
   * Comienza a fusionar los subarrays de tamaño 1 en orden creciente según el género de las películas.
   * Para esto, se comparan los géneros de los elementos de los subarrays izquierdo y derecho, insertando el menor en el array principal.
   * Este proceso continúa hasta que todos los elementos han sido fusionados.

**Clasificación del algoritmo**: Divide y vencerás. Divide el array en mitades, resuelve recursivamente cada mitad y combina las soluciones para obtener el resultado final.Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

**Cálculo del T(n) para mergersort (agrupar por genero).**

Merge =

Mergesort =

Ecuación de recurrencia = ,

Patrón:

Sustitución:

esto para el mergesort, dentro de la función de agrupar por genero hacemos una llamada a mergesort y además usamos un for para iterar sobre todas las películas que tenemos eso dejando que es y como sigue siendo menor a del mergesort se puede decir que la función de group\_by\_genre , cumpliendo el requisito de que sea de complejidad , eso siendo para el peor caso, y en ese caso no hay mejor caso, ya que el mergesort siempre se ejecutara estén o no estén ordenado los datos, y la impresión siempre ser porque siempre recorrerá todo igualmente, entonces podemos decir que en todos los casos da la misma complejidad.

# **Top 10 películas más populares por calificación**

**Complejidad requerida: O(nlogn)**

Para esta funcionalidad, decidimos utilizar el algoritmo **HeapSort** debido a su eficiencia en la clasificación de grandes volúmenes de datos y a su capacidad para seleccionar de manera eficiente los elementos más importantes. HeapSort garantiza una complejidad temporal de O(nlogn) en todos los casos (mejor, peor y promedio), lo que lo hace adecuado para ordenar las películas según su calificación.

**Descripción del algoritmo utilizado**

HeapSort es un algoritmo de clasificación que sigue el paradigma de **Transforma y Vence**. Primero, transforma el array original en una estructura de montículo binario máximo, lo que facilita la extracción iterativa del elemento más grande. Posteriormente, vence al problema utilizando el montículo para obtener los elementos ordenados de forma eficiente.

El algoritmo consta de dos etapas principales:

1. **Construcción del montículo**: Se reorganizan los elementos del array para formar un montículo máximo, en el que cada nodo es mayor que sus hijos.
2. **Ordenación**: Se intercambia la raíz del montículo (el elemento más grande) con el último elemento del array, se reduce el tamaño del montículo y se reorganiza nuevamente. Este proceso se repite hasta que el array esté completamente ordenado.

**Justificación de la elección del algoritmo**

Elegimos HeapSort por las siguientes razones:

* **Eficiencia**: Tiene una complejidad temporal de O(nlogn) lo que lo hace adecuado para procesar grandes conjuntos de datos, como una lista extensa de películas.
* **Espacio constante**: A diferencia de otros algoritmos como MergeSort, HeapSort no requiere memoria adicional significativa para almacenar subarrays, ya que trabaja directamente sobre el array original.
* **Orden parcial**: Aunque HeapSort ordena todo el array, el top 10 de las películas más populares puede extraerse directamente tras ordenar, ya que los elementos con mayor calificación se colocan al principio del array.

**Explicación del funcionamiento del algoritmo**

1. **Transformación al montículo**:
   * El array original se reorganiza para formar un montículo máximo, garantizando que cada nodo sea mayor que sus hijos.
   * Este proceso toma O(n)), ya que cada nodo se ajusta en un tiempo proporcional a su profundidad (O(logn)
2. **Extracción iterativa y ordenación**:
   * Se extrae el elemento más grande (la raíz del montículo) y se intercambia con el último elemento del array.
   * Luego, se ajusta el montículo restante para mantener la propiedad del montículo máximo. Este proceso se repite n veces, y cada reconstrucción del montículo toma O(logn).

**Clasificación del algoritmo**: Transforma y vencerás. Transforma el array en un montículo para luego vencer el problema utilizando su estructura jerárquica.

Texto

Descripción generada automáticamente

# **Películas ordenadas por calificación (1-5 estrellas), de mayor a menor.**

**Complejidad requerida: MENOR a O(nlogn).**

Para implementar la ordenación de películas por calificación, elegimos el algoritmo **Counting Sort**, que es eficiente para datos con un rango limitado de valores, como en este caso, donde las calificaciones de las películas están entre 1 y 5 estrellas. Este algoritmo cumple con el requisito de complejidad menor a O(nlogn) y es ideal para ordenar rápidamente listas con valores discretos.

**Descripción del algoritmo utilizado**

El algoritmo **Counting Sort** cuenta la cantidad de veces que aparece cada calificación y luego reconstruye la lista ordenada de acuerdo con esas frecuencias. Debido a su naturaleza, es especialmente útil cuando los datos a ordenar tienen un rango limitado.

Consta de tres etapas principales:

**1. Determinación del rango de calificaciones**

Se recorre el arreglo de películas para encontrar el valor mínimo y máximo de calificación.

**Complejidad:**  
O(n) donde n es el número de películas.

**2. Construcción del arreglo de conteo**

Se crea un arreglo de conteo que guarda la frecuencia de cada calificación (en este caso, entre 1 y 5). Luego, se convierte este arreglo de frecuencias en un arreglo acumulado, de modo que cada índice indique la posición de la calificación en el arreglo ordenado.

**Complejidad:**  
O(k), donde k es el rango de las calificaciones (1 a 5).

**3. Reconstrucción del arreglo ordenado**

Usando el arreglo de conteo acumulado, se coloca cada película en su posición ordenada dentro de un nuevo arreglo.

**Complejidad:**  
O(n), donde n es el número de películas.

**Justificación de la elección del algoritmo**

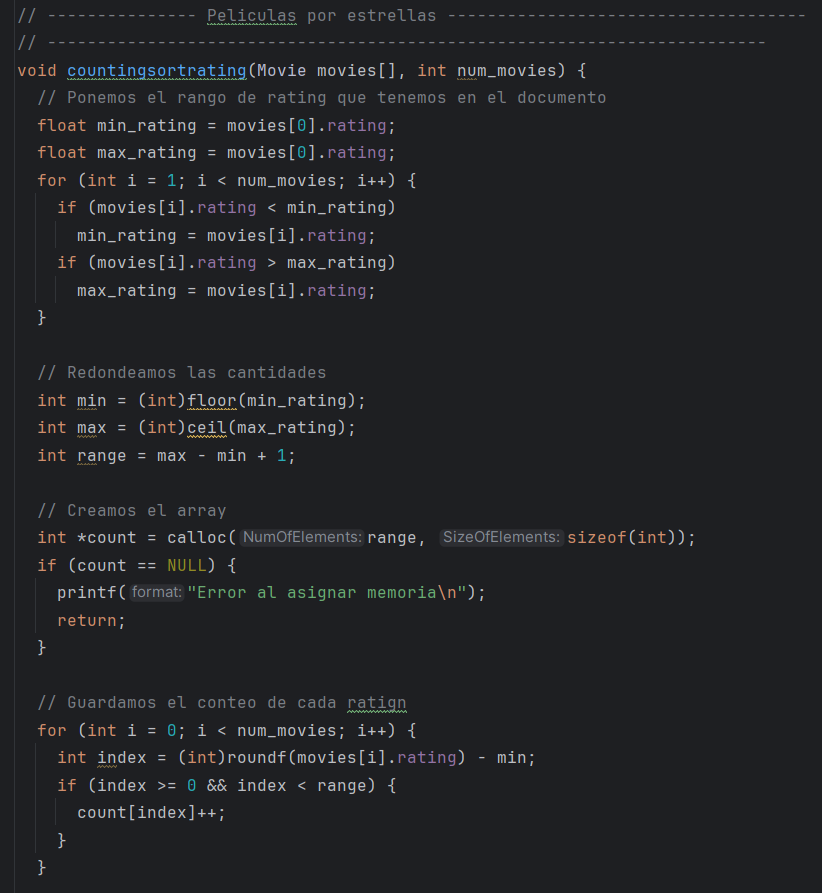
Elegimos **Counting Sort** por las siguientes razones:

* **Eficiencia:** Su complejidad es O(n+k) lo que es mucho más rápido que métodos de ordenamiento basados en comparaciones como **QuickSort** o **MergeSort**, que tienen una complejidad de O(nlogn)
* **Simplicidad y adecuación:** Como el rango de calificaciones es pequeño (de 1 a 5), este algoritmo es ideal para ordenar las películas de manera rápida y eficiente.

**Explicación del funcionamiento del algoritmo**

1. **Determinación del rango de calificaciones:**  
   Se recorre el arreglo de películas para encontrar la calificación mínima y máxima. Este paso es necesario para construir el arreglo de conteo correctamente.
2. **Construcción del arreglo de conteo:**  
   Se crea un arreglo de frecuencias para cada calificación posible (de 1 a 5). Posteriormente, se transforma este arreglo de frecuencias en un arreglo acumulado, lo que nos permite determinar la posición final de cada calificación en el arreglo ordenado.
3. **Reconstrucción del arreglo ordenado:**  
   Usando el arreglo de conteo acumulado, se reconstruye el arreglo de películas, colocando cada película en la posición correspondiente de acuerdo con su calificación.

**Clasificación del algoritmo:** Transforma y vencerás.



Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

# **Cantidad de películas que se pueden ver para acumular un tiempo exacto.**

**Complejidad requerida: menor a O(n3).**

**Descripción del algoritmo utilizado**

La solución que buscamos se basa en **programación dinámica (DP)**, específicamente en una variante del problema de la mochila. Este enfoque divide el problema en subproblemas más pequeños, construyendo una tabla que indica si es posible alcanzar un tiempo exacto utilizando un subconjunto de películas.

**Fases del algoritmo:**

1. **Definición de estado:**  
   Utilizamos una tabla dp[i][j] donde:
   * dp[i][j] = 1 indica que es posible alcanzar un tiempo total j utilizando las primeras i películas.
   * dp[i][j] = 0 en caso contrario.
2. **Inicialización:**
   * dp[0][0] = 1: Es posible alcanzar un tiempo de 0 minutos sin seleccionar películas.
   * dp[0][j] = 0 para j>0 : No es posible alcanzar un tiempo positivo sin películas.
3. **Transición:**  
   Para cada película i y cada tiempo j:
   * Si no seleccionamos la película i: dp[i][j] = dp[i-1][j].
   * Si seleccionamos la película i: dp[i][j] = dp[i-1][j - movies[i-1].time] (si j≥movies[i−1].time).
4. **Reconstrucción:**  
   Si dp[n][t] = 1, reconstruimos las películas seleccionadas rastreando las decisiones tomadas durante la construcción de la tabla.

**Justificación de la elección del algoritmo**

1. **Eficiencia:**  
   Este enfoque tiene una complejidad temporal de O(n⋅t), mucho menor que el enfoque inicial de fuerza bruta O(2n) y cumple con el requisito de O(n3)
2. **Adecuación:**  
   La programación dinámica es ideal para problemas de combinatoria, como encontrar subconjuntos de películas que sumen exactamente el tiempo objetivo.
3. **Simplicidad en la reconstrucción:**  
   La tabla dp permite rastrear las películas seleccionadas sin necesidad de estructuras auxiliares complejas.

**Explicación del funcionamiento del algoritmo**

1. **Construcción de la tabla DP:**  
   Iteramos sobre cada película i y cada tiempo j, actualizando dp[i][j] según las dos posibles decisiones (seleccionar o no seleccionar la película).  
   Complejidad: O(n⋅t) donde n es el número de películas y t el tiempo objetivo.
2. **Reconstrucción de la solución:**  
   Si dp[n][t] = 1, iteramos hacia atrás en la tabla para identificar qué películas contribuyeron al tiempo total. Esto requiere O(n)

**Clasificación del algoritmo:** Programación dinámica.

**Texto

Descripción generada automáticamente**

# **Búsqueda por título o palabras clave**

**Complejidad requerida: MENOR a O(n2)**

Para implementar la funcionalidad de búsqueda de títulos o palabras clave, elegimos el **algoritmo Knuth-Morris-Pratt (KMP)**, un algoritmo eficiente para la búsqueda de patrones en cadenas de texto. Este algoritmo tiene una complejidad menor a O(n2), cumpliendo con el requisito establecido, y es ideal para búsquedas rápidas y múltiples en una lista de películas.

**Descripción del algoritmo utilizado**

El algoritmo KMP es un método eficiente que evita comparaciones redundantes al buscar un patrón dentro de un texto, utilizando una **tabla de prefijos** para identificar la posición más adecuada para reiniciar la comparación después de un desajuste.

Consta de dos etapas principales:

1. **Construcción de la tabla de prefijos**:
   * Esta tabla indica la longitud del prefijo que coincide con el sufijo hasta una posición dada en el patrón.
   * La tabla permite reiniciar las comparaciones en el texto sin retroceder en el índice del texto, lo que optimiza el proceso.
   * Complejidad: O(m) donde m es la longitud del patrón.
2. **Búsqueda con el algoritmo KMP**:
   * El algoritmo recorre el texto y utiliza la tabla de prefijos para ajustar el índice del patrón tras un desajuste.
   * Esto evita comparaciones innecesarias y asegura una complejidad lineal con respecto al tamaño del texto O(n) donde n es la longitud del texto.

**Justificación de la elección del algoritmo**

Elegimos el algoritmo KMP por las siguientes razones:

* **Eficiencia**: Tiene una complejidad temporal lineal para una sola búsqueda (O(n+m) lo que lo hace más eficiente que enfoques como fuerza bruta (O(n⋅m).
* **Reutilización**: Al realizar búsquedas en múltiples títulos de películas, el algoritmo es capaz de mantener su rendimiento gracias a la optimización que evita retrocesos innecesarios.
* **Idoneidad para cadenas largas**: Como los títulos de películas y las palabras clave pueden variar en longitud, KMP se adapta bien a estos escenarios.

**Explicación del funcionamiento del algoritmo**

1. **Construcción de la tabla de prefijos**:
   * Iteramos sobre el patrón y calculamos la longitud del prefijo más largo que coincide con el sufijo hasta cada posición.
   * Este paso toma O(m), donde m es la longitud del patrón.
2. **Búsqueda en el texto**:
   * Recorremos el texto y el patrón simultáneamente.
   * Cuando ocurre un desajuste, usamos la tabla de prefijos para reiniciar la comparación desde la posición adecuada del patrón, sin retroceder en el texto.
   * Este paso toma O(n), donde n es la longitud del texto.
3. **Búsqueda en múltiples títulos**:
   * El algoritmo KMP se aplica a cada título de película, que tiene una longitud promedio de n, con un patrón de longitud m.
   * Para una lista de k películas, la complejidad total es O(k⋅(n+m), lo que es significativamente más eficiente que métodos cuadráticos (O(k⋅n2).

**Clasificación del algoritmo: Divide y Vencerás (subdivisión implícita)**. Aunque no es un algoritmo clásico de este paradigma, KMP divide implícitamente el texto en subproblemas al usar la tabla de prefijos para manejar desajustes.

**Texto

Descripción generada automáticamente**

**Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente**

# **Historial películas.**

**Complejidades requeridas:**

* **Agregar al historial**: O(1).
* **Eliminar la primera película vista**: O(1)
* **Visualizar todo el historial**: O(n)

Para implementar el historial de visualización, decidimos utilizar una **cola.** Este tipo de estructura de datos es ideal para manejar operaciones que requieren acceso en orden FIFO como mantener un registro secuencial de visualizaciones de películas.

**Descripción del algoritmo utilizado**

La **cola** se implementa con nodos dinámicos que contienen información sobre la película y apuntan al siguiente nodo. Incluye tres operaciones principales:

1. **Agregar al historial (enqueue).**
   * Se añade un nodo al final de la cola, actualizando el puntero del último nodo existente o el inicio si la cola está vacía.
   * La operación toma tiempo constante O(1) ya que solo modifica un par de punteros.
2. **Eliminar la primera película (dequeue)**:
   * Se elimina el nodo al frente de la cola y se actualiza el puntero de inicio.
   * Si el historial queda vacío, se actualiza también el puntero final.
   * La operación toma tiempo constante O(1) porque solo involucra cambios en los punteros del nodo inicial.
3. **Mostrar todo el historial**:
   * Se recorren todos los nodos de la cola para imprimir sus detalles.
   * La operación tiene una complejidad de O(n) ya que debe visitar cada nodo de la estructura.

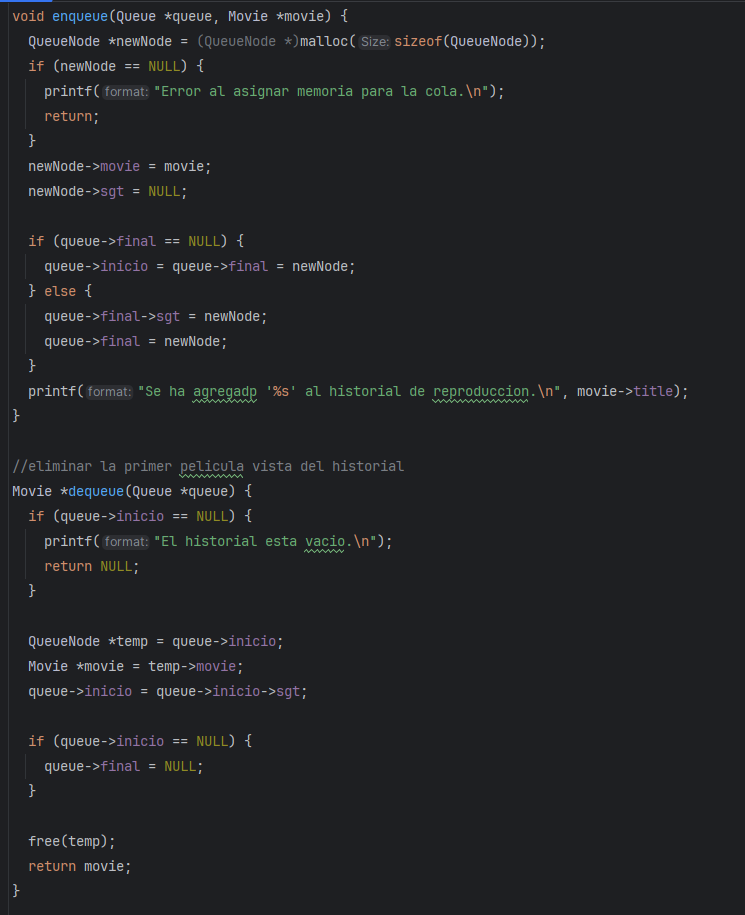
**Justificación de la elección del algoritmo**

1. **Eficiencia de las operaciones principales**:
   * Las operaciones de insertar y eliminar en una cola están diseñadas para ejecutarse en O(1). lo cual es crucial para un historial dinámico que se actualiza constantemente.
2. **Orden natural**:
   * La cola respeta el orden cronológico de visualización, manteniendo las películas en el orden en que fueron vistas.
3. **Simplicidad de implementación**:
   * Una cola se puede implementar fácilmente utilizando estructuras dinámicas con punteros, manteniendo la flexibilidad para crecer según sea necesario.

**Explicación del funcionamiento del algoritmo**

1. **Inicialización**:
   * Se crea una cola vacía con punteros inicio y final establecidos en NULL.
2. **Agregar una película al historial**:
   * Se asigna memoria para un nuevo nodo, se almacena la información de la película y se ajustan los punteros para que el nuevo nodo sea el último de la cola.
3. **Eliminar la primera película vista**:
   * Se elimina el nodo al frente de la cola, liberando su memoria y actualizando los punteros para reflejar el siguiente nodo como inicio. Si el historial queda vacío, se establece final en NULL.
4. **Mostrar todo el historial**:
   * Se recorre desde el nodo inicial hasta el último, imprimiendo los detalles de cada película almacenada en los nodos.

**Clasificación del algoritmo**: Algoritmo de estructura de datos dinámica.



Texto

Descripción generada automáticamente

**Cálculo de T(n) para todo lo relacionado con el historial.**

Agregar al historial. complejidad requerida.

Para este se usó el método enqueue sobre la queue del historial, al no usar ningún ciclo ni llamada recursiva todo se hace de forma constante, se asigna memoria para el nodo de la película = , se actualiza punteros = y se manda el mensaje de confirmación de que se agregó la película = , todo se traduce a .

Entonces siendo este el mismo caso para todos, ya que no hay mejor caso, quedando entonces que en el peor caso y en el caso promedio .

Visualizar y eliminar la primera película en el historial. Complejidad requerida:.

En este se usó el método dequeue sobre la queue del historial, como en la anterior al no usar ningún ciclo o llamada recursiva todo se hace de forma constante, en esta primero se verifica que la cola no este vacía = , después se actualiza el puntero del inicio = , se libera el nodo quitado = y se retorna ese nodo = , como en la anterior al no usar ningún ciclo o llamada recursiva se traduce a .

Igualmente, en esta no hay peor ni mejor caso, es el mismo caso para todos, entonces 𝜖 Ο1 para el peor caso, en el mejor caso y en el caso promedio .

Mostrar todo el historial de reproducción. Complejidad requerida:.

En esta función se usó una función para imprimir todos los elementos de la queue, se hace a través de un bucle for por ende el tiempo de ejecución quedará en , sin importar que solo haya 1, 2 o hasta n elementos, siempre será en torno a n, y por ende siempre se recorrerá toda la estructura.

Por lo tanto, todos los casos de complejidad son iguales, entonces tenemos que en el peor de los casos, y para el mejor caso es lo mismo que y en el caso promedio más de lo mismo .