



**udp** UNIVERSIDAD  
DIEGO PORTALES

UNIVERSIDAD DIEGO PORTALES  
ESCUELA DE INFORMÁTICA &  
TELECOMUNICACIONES

ESTRUCTURA DE DATOS &  
ALGORITMOS

---

## Laboratorio 3: Algoritmos de Ordenamiento y Búsqueda

---

*Autores:*

*Cinthya Fuentealba Bravo*

*Ignacia Reyes Ojeda*

*Profesor: Cristian Llull Torres*

22 de Octubre 2025

---

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Implementación</b>	<b>3</b>
2.1. Estructura General . . . . .	3
2.2. Clase <i>Movie</i> . . . . .	3
2.3. Clase <i>MovieCatalog</i> . . . . .	5
2.4. Algoritmos de ordenamiento implementados . . . . .	7
2.5. Métodos de Búsqueda . . . . .	12
2.6. clase <i>CsvIMDBLoader</i> . . . . .	21
2.7. Clase Main . . . . .	21
<b>3. Experimentación</b>	<b>21</b>
3.1. Experimento 1 . . . . .	21
3.2. Experimento 2 . . . . .	25
<b>4. Conclusión</b>	<b>28</b>

---

# 1. Introducción

En la actualidad, existen múltiples casos o situaciones en donde se debe trabajar sobre grandes volúmenes de información, por ende, es necesaria la utilización de herramientas que nos permitan manipular los datos de una manera eficiente. Entre las principales acciones que se debe realizar para trabajar con grandes volúmenes de datos, es fundamental organizar y clasificar la información para poder trabajar sobre ellos de la mejor manera posible.

Al analizar el área del entretenimiento digital, plataformas como por ejemplo, *Netflix*, *Prime Video*, *Disney+*, entre otras; se puede apreciar que dichas plataformas deben gestionar extensos catálogos de películas y series, los cuales deben de tener la posibilidad de ordenar y filtrar por diferentes criterios, según la preferencia del usuario. Sin embargo, no solo se almacenan películas o series, sino también manejan otra información igual de importantes como lo son: el título, elenco, descripción de la trama, la valoración de los usuarios, año de lanzamiento, director, género, entre muchos otras atributos que definen y hacen única cada creación cinematográfica.

Detrás de la experiencia fluida y personalizada que percibe un usuario, es fundamental la presencia de algoritmos de ordenamiento y búsqueda que sean capaces de procesar grandes cantidades de información en el menor tiempo posible. Este laboratorio tiene como objetivo aplicar conceptos de ordenamiento y búsqueda en un contexto práctico, implementando un sistema de gestión de películas basado en el dataset “*IMDb Top 1000 Movies and TV Shows*” de Kaggle.

Dentro del desarrollo práctico mediante la implementación en Java es posible realizar un orden y categorización sobre las películas, mediante diferentes métodos de ordenamiento principalmente. Para ello, se implementó la creación de dos clases fundamentales las cuales son: *clase Movie* y *clase MovieCatalog*.

Además de la creación de un código informático es importante entender el funcionamiento de estos métodos de ordenamiento, ya que funcionan de diferentes formas. También se realiza el análisis del algoritmo y los métodos de búsqueda lineal y binaria, comprobando su rendimiento en un gran volumen de datos. Al desarrollar este laboratorio es posible comprender la importancia de la elección del algoritmo adecuado según el contexto en el que se encuentre, entender la complejidad tanto algorítmica como de estructura, y poder comparar la eficiencia de dichos métodos.

En el siguiente repositorio de GitHub se encuentra la información completa de este laboratorio: [https://github.com/Cinthya982012/Laboratorio\\_3\\_EDA\\_2025\\_2](https://github.com/Cinthya982012/Laboratorio_3_EDA_2025_2). En el enlace se encuentra, el código .Java y el archivo L<sup>A</sup>T<sub>E</sub>X para su análisis.

---

## 2. Implementación

### 2.1. Estructura General

Este laboratorio fue realizado en lenguaje Java, se utilizaron estructuras como `ArrayList` para poder gestionar los datos. Se compone principalmente de dos clases fundamentales: `class Movie` y `class MovieCatalog`, además se creó una clase auxiliar para poder cargar el archivo CSV a la cual llamamos la clase: `class CsvIMDBLoader`, y finalmente la infaltable: `class Main` implementada para la ejecución de pruebas y experimentos, que se muestran por consola. Las clases implementadas se muestran a continuación en este informe.

### 2.2. Clase *Movie*

Esta clase es la encargada de representar una película individual, dentro del catálogo y también con la información relevante solicitada.

La clase está formada por los siguientes atributos:

- `String title` : corresponde al título (nombre) de la película de tipo `String`.
- `String director`: corresponde al nombre del director que dirigió dicha filmación, de tipo `String`.
- `String genre`: corresponde al género o categoría en el cual se clasifica la película, de tipo `String`.
- `int releaseYear`: representa el año en el cual se estrenó la película, de tipo `int`.
- `double rating`: representa la calificación promedio de los usuarios o la crítica sobre la película. Este valor es de tipo `double` y se encuentra entre los valores de 0.0 y 10.0.

La clase también está formada por los siguientes métodos:

- `Movie(String title, String director, String genre, int releaseYear, double rating)`: corresponde al constructor que inicializa los atributos de la película.
- **Setters y Getters**: métodos encargados de obtener y asignar cada atributo.
- `public static boolean doubleEquals(double a, double b)`: encargado de comparar dos valores `double` considerando una tolerancia EPS de  $\pm 0.001$ .
- `public String toString()`: encargado de imprimir información de la película por consola.

A continuación se presenta la clase implementada:

---

```

1 //CLASE MOVIE
2 class Movie{
3     //Atributos clase Movie
4     private String title; // Titulo de la pelicula
5     private String director; // Nombre del director de la pelicula
6     private String genre; // Genero o categoria de la pelicula
7     private int releaseYear; // A o de lanzamiento de la pelicula
8     private double rating; // Clasificacion promedio de la
        pelicula (entre 0 y 10)
9
10    public static final double EPS = 0.0001; //Tolerancia para
        comparar doubles
11
12    //Metodos clase Movie
13
14    //Constructor
15    public Movie(String title, String director, String genre, int
        releaseYear, double rating){
16        this.title = title;
17        this.director = director;
18        this.genre = genre;
19        this.releaseYear = releaseYear;
20        this.rating = rating;
21    }
22
23    //Setters y Getters
24    String setTitle(){return title;}
25    String setDirector(){return director;}
26    String setGenre(){return genre;}
27    int setReleaseYear(){return releaseYear;}
28    double setRating(){return rating;}
29
30
31    public String getTitle(){return title;}
32    public String getDirector(){return director;}
33    public String getGenre(){return genre;}
34    public int getReleaseYear(){return releaseYear;}
35    public double getRating(){return rating;}
36
37    //Compara doubles con tolerancia
38    public static boolean doubleEquals(double a, double b){
39        return Math.abs(a - b) <= EPS;
40    }
41
42    //Para imprimir
43    public String toString() {
44        return
45            "Pelicula:\n" + "Titulo: " + title + "\n" +
                "Director: " + director + "\n" + "Genero = " +
                genre + "\n" + "A o de estreno: " +
                releaseYear + "\n" + "Calificacion: " + rating
                + "\n";}}

```

---

## 2.3. Clase *MovieCatalog*

Esta clase esta encargada de gestionar un conjunto de objetos de tipo `Movie`, permitiendo su ordenamiento y búsqueda mediante diferentes algoritmos, es decir, corresponde al catalogo de todas las películas.

Esta clase esta formada por os siguientes atributos:

- `ArrayList<Movie>movies`: corresponde a una lista de películas del catálogo.
- `String sortedByAttribute`: encargado de indicar el parámetro por el cual se encuentra ordenado actualmente el catálogo, pueden ser: `rating`, `genre`, `director`, `year`, o `null` en caso de estar vacía.

También se implementaron los siguientes métodos:

- `MovieCatalog(ArrayList<Movie>movies)`: es un constructor que inicializa los atributos de lista de películas. Se inicializa el atributo `sortedByAttribute` en nulo.
- `public int size()`: retorna el tamaño de la lista de películas.
- `public boolean isEmpty()`: verifica si la lista de películas está vacía o no.
- `public String getSortedByAttribute()`: retorna el atributo `sortedByAttribute`.
- `public void agregarPelicula(Movie m)`: agrega una película a la lista y marca el catálogo como desordenado, ya que se pierde el orden.
- `private String actualizarAtributo(String cualidad)`: normaliza el nombre del atributo recibido (`rating`, `genre`, `director`, `year`) a un atributo válido. Por defecto retorna `rating`, por indicación en guía.
- `private Comparator<Movie>comparadorPor(String atributo)`: devuelve un `Comparator<Movie>` según el atributo seleccionado.
- `public void sortByAlgorithm(String algorithm, String attribute)`: Ordena la lista de películas según el algoritmo y atributo indicados. Soporta: `insertion`, `selection`, `merge`, `quick`, `radix` (solo `year`), y por defecto `Collections.sort`.

La implementación se muestra a continuación:

```
1 //CLASE MOVIE CATALOG
2 class MovieCatalog {
3     private ArrayList<Movie> movies; // Lista de peliculas
4     private String sortedByAttribute; //Atributo por el cual esta
      ordenada la lista
5
6     public MovieCatalog(ArrayList<Movie> movies) {
7         if (movies == null) {
8             this.movies = new ArrayList<>();
9         } else {
```

---

```

10         this.movies = new ArrayList<>(movies);
11     }
12     this.sortedByAttribute = null;
13 }
14
15 public int size() {
16     return movies.size();
17 }
18
19 public boolean isEmpty() {
20     return movies.isEmpty();
21 }
22
23 public ArrayList<Movie> obtenerPeliculas() {
24     return new ArrayList<>(movies);
25 }
26
27 public String getSortedByAttribute() {
28     return sortedByAttribute;
29 }
30
31 public void agregarPelicula(Movie m) {
32     movies.add(m);
33     sortedByAttribute = null;
34 }
35
36 private String actualizarAtributo(String cualidad) {
37     if (cualidad == null) return "rating";
38     String a = cualidad.trim().toLowerCase();
39     if (a.equals("rating") || a.equals("genre") ||
40         a.equals("director") || a.equals("year")) {
41         return a;
42     }
43     return "rating";
44 }
45
46 private Comparator<Movie> comparadorPor(String atributo) {
47     switch (atributo) {
48         case "rating":
49             return
50                 Comparator.comparingDouble(Movie::getRating);
51         case "genre":
52             return Comparator.comparing(m ->
53                 m.getGenre().toLowerCase());
54         case "director":
55             return Comparator.comparing(m ->
56                 m.getDirector().toLowerCase());
57         case "year":
58             return
59                 Comparator.comparingInt(Movie::getReleaseYear);
60         default:
61             return

```

---

```

57         Comparator.comparingDouble(Movie::getRating);
58     }
59
60     public void sortByAlgorithm(String algorithm, String
61         attribute) {
62         String atributo = actualizarAtributo(attribute);
63         String algoritmo = (algorithm == null) ? " " :
64             algorithm.trim().toLowerCase();
65         Comparator<Movie> comparador = comparadorPor(atributo);
66
67         switch (algoritmo) {
68             case "insertion":
69                 insertionSort(comparador);
70                 break;
71             case "selection":
72                 selectionSort(comparador);
73                 break;
74             case "merge":
75                 mergeSort(comparador);
76                 break;
77             case "quick":
78                 quickSort(comparador);
79                 break;
80             case "radix":
81                 if("year".equals(atributo)) {
82                     radixSortYear();
83                 } else {
84                     mergeSort(comparador);
85                 }
86                 break;
87             default:
88                 Collections.sort(movies, comparador);
89         }
90         sortedByAttribute = atributo;
91     }
92 }

```

## 2.4. Algoritmos de ordenamiento implementados

En el desarrollo se implementaron diferentes algoritmos de ordenamiento, con el objetivo de comparar sus rendimientos y complejidades, estos se muestran a continuación:

### 1. *InsertionSort*

Este algoritmo recorre el arreglo elemento por elemento, insertando cada película en la posición correspondiente de la parte ya ordenada de la lista.

Su complejidad temporal es  $O(n^2)$ , lo que lo hace eficiente solo para conjuntos de datos pequeños o casi ordenados.



A continuación el algoritmo implementado:

```
1      //Insertion Sort
2      public void insertionSort(Comparator<Movie>
3          comparador) {
4          for (int i = 1; i < movies.size(); i++) {
5              Movie clave = movies.get(i);
6              int j = i - 1;
7
8              // Mueve elementos que sean mayores que 'clave'
9              // hacia la derecha
10             while (j >= 0 && comparador.compare(movies.get(j),
11                 clave) > 0) {
12                 movies.set(j + 1, movies.get(j));
13                 j--;
14             }
15             // Inserta la clave en la posición correcta
16             movies.set(j + 1, clave);
17         }
18     }
```

## 2. *MergeSort*

Funciona con la frase “divide y vencerás”, ya que divide recursivamente la lista en mitades hasta llegar a listas de un solo elemento, y luego las combina en orden.

Su complejidad temporal promedio es  $O(n \log n)$ , lo que lo convierte en uno de los algoritmos más eficientes para grandes volúmenes de datos.

La implementación del algoritmo a continuación:

```
1      //Merge Sort
2      public void mergeSort(Comparator<Movie> comparador) {
3          if (movies.size() <= 1) {return;}
4          ArrayList<Movie> aux = new ArrayList<>(movies);
5          mergeSortRec(0, movies.size()-1, aux, comparador);
6      }
7
8      private void mergeSortRec(int lo, int hi, ArrayList<Movie>
9          aux, Comparator<Movie> comparador) {
10         if(lo >= hi) {return;}
11         int mid = lo + (hi - lo) / 2;
12         mergeSortRec(lo, mid, aux, comparador);
13         mergeSortRec(mid + 1, hi, aux, comparador);
14         merge(lo, mid, hi, aux, comparador);
15     }
16
17     private void merge(int lo, int mid, int hi,
18         ArrayList<Movie> aux, Comparator<Movie> comparador) {
19         for(int t = lo; t <= hi; t++) {
20             aux.set(t, aux.get(t));
21         }
22         int i = lo;
```

```

21     int j = lo;
22     int k = mid + 1;
23
24     while(j <= mid && k <= hi) {
25         if (comparador.compare(aux.get(j), aux.get(k)) <=
26             0) {
27             movies.set(i++, aux.get(j++));
28         }
29         else {
30             movies.set(i++, aux.get(k++));
31         }
32     }
33     while(j <= mid) {
34         movies.set(i++, aux.get(j++));
35     }

```

### 3. *RadixSort*

Ordena los elementos numéricos (en este caso, el atributo *releaseYear*) considerando cada dígito individual. Este método no realiza comparaciones directas, por lo que su rendimiento depende del rango de los datos y no de su cantidad.

Su complejidad promedio es  $O(n \cdot k)$ , donde  $k$  representa el número de dígitos o posiciones analizadas.

El siguiente algoritmo corresponde al implementado:

```

1  //Radix Sort
2  public void radixSortYear() {
3      if (movies.isEmpty()) return;
4
5      int min = Integer.MAX_VALUE, max = Integer.MIN_VALUE;
6      for (int i = 0; i < movies.size(); i++) {
7          Movie m = movies.get(i);
8          int y = m.getReleaseYear();
9          if (y < min) min = y;
10         if (y > max) max = y;
11     }
12     int ajuste;
13     if(min < 0){
14         ajuste = -min;
15     }
16     else{
17         ajuste = 0;
18     }
19
20     int[] llave = new int[movies.size()];
21     for (int i = 0; i < movies.size(); i++) {
22         llave[i] = movies.get(i).getReleaseYear() + ajuste;
23     }
24
25     int posicionDigito = 1;

```

```

26     ArrayList<Movie> temporal = new
        ArrayList<>(Collections.nCopies(movies.size(),
        (Movie) null));
27     while ((max + ajuste) / posicionDigito > 0) {
28         int[] contador = new int[10];
29
30         for (int k : llave) {contador[(k / posicionDigito)
            % 10]++;}
31
32         for (int i = 1; i < 10; i++) {contador[i] +=
            contador[i - 1];}
33
34         for (int i = movies.size() - 1; i >= 0; i--) {
35             int digit = (llave[i] / posicionDigito) % 10;
36             int pos = --contador[digit];
37             temporal.set(pos, movies.get(i));
38         }
39
40         for (int i = 0; i < movies.size(); i++) {
41             movies.set(i, temporal.get(i));
42             llave[i] = movies.get(i).getReleaseYear() +
                ajuste;
43         }
44         posicionDigito *= 10;
45     }
46 }

```

Además se realizaron dos algoritmos más de ordenamiento, estos fueron:

#### 4. *SelectionSort*

Este algoritmo busca en cada iteración el elemento más pequeño y lo coloca en la posición correspondiente. Tiene una complejidad de  $O(n^2)$ .

A continuación el algoritmo:

```

1     //Selection Sort
2     public void selectionSort(Comparator<Movie> comparador) {
3         for (int i = 1; i < movies.size(); i++) {
4             //Indice minimo
5             int indiceMin = i;
6             for (int j = i + 1; j < movies.size(); j++) {
7                 if (comparador.compare(movies.get(j),
                    movies.get(indiceMin)) < 0) {
8                     indiceMin = j;
9                 }
10            }
11            if (indiceMin != i) {
12                Movie aux = movies.get(i);
13                movies.set(i, movies.get(indiceMin));
14                movies.set(indiceMin, aux);
15            }
16        }
17    }

```

## 5. *QuickSort*

Este método de ordenamiento selecciona un pivote y luego organiza los elementos menores a un lado y los mayores al otro lado. Su complejidad es de  $O(n \log n)$ . En la práctica es uno de los más rápidos para grandes cantidades de datos.

A continuación la implementación:

```
1      //Quick Sort
2      public void quickSort(Comparator<Movie> comparador) {
3          if (movies.size() <= 1) {return;}
4
5          Collections.shuffle(movies);
6          quickRec(0, movies.size() - 1, comparador);
7      }
8
9      private void quickRec(int lo, int hi, Comparator<Movie>
10         comparador) {
11          if (lo >= hi) {return;}
12          int p = particionar(lo, hi, comparador);
13          quickRec(lo, p - 1, comparador);
14          quickRec(p + 1, hi, comparador);
15      }
16
17      private int particionar(int lo, int hi, Comparator<Movie>
18         comparador) {
19          Movie pivot = movies.get(hi);
20          int i = lo;
21          for (int j = lo; j < hi; j++) {
22              if (comparador.compare(movies.get(j), pivot) <= 0)
23              {
24                  swap(i, j);
25                  i++;
26              }
27          }
28          swap(i, hi);
29          return i;
30      }
31
32      private void swap(int i, int j) {
33          if (i == j) return;
34          Movie tmp = movies.get(i);
35          movies.set(i, movies.get(j));
36          movies.set(j, tmp);
37      }
```

Una vez que se encuentra la lista ordenada, el atributo `sortedByAttribute` se actualiza como ordenado.

---

## 2.5. Métodos de Búsqueda

Otros de los métodos necesarios para este laboratorio es la utilización de métodos de búsqueda. Los métodos aplicados fueron:

- `public ArrayList<Movie>getMoviesByRating(double rating)`: encargado de retornar una lista de películas según la calificación que se busca, esta debe ser igual al parámetro entregado.

Su implementación:

```
1      public ArrayList<Movie> getMoviesByRating(double
2          rating) {
3          if ("rating".equals(sortedByAttribute)) {
4              //Busqueda Binaria (necesita el catalogo
5              //previamente ordenado por rating ascendente)
6              return buscarPorRatingBinaria(rating);
7          } else {
8              //Busqueda lineal
9              ArrayList<Movie> listaRating = new ArrayList<>();
10             for (int i = 0; i < movies.size(); i++) {
11                 Movie m = movies.get(i);
12                 if (Movie.doubleEquals(m.getRating(), rating))
13                     {
14                         listaRating.add(m);
15                     }
16             }
17             return listaRating;
18         }
19     }
20
21     //Busqueda Binaria para rating (lista ordenada previamente)
22     private ArrayList<Movie> buscarPorRatingBinaria(double
23         rating) {
24         ArrayList<Movie> listaRating = new ArrayList<>();
25         if (movies.isEmpty()) {
26             return listaRating;
27         }
28
29         int lo = limiteInferiorRating(rating);
30         int hi = limiteSuperiorRating(rating) - 1;
31
32         if (lo <= hi && lo >= 0 && hi < movies.size()) {
33             for (int i = lo; i <= hi; i++) {
34                 listaRating.add(movies.get(i));
35             }
36         }
37         return listaRating;
38     }
39
40     // Primer indice con rating >= (rating - EPS)
41     private int limiteInferiorRating(double rating) {
42         double objetivo = rating - Movie.EPS;
```

```

39         //Comienzo
40         int l = 0;
41
42         //Final
43         int rr = movies.size();
44
45         //Mientras l sea menor que rr entra al ciclo
46         while (l < rr) {
47             //Se define la mitad
48             int mid = l + (rr - l) / 2;
49             double valor = movies.get(mid).getRating();
50             if (valor < objetivo) {
51                 l = mid + 1;
52             } else {
53                 rr = mid;
54             }
55         }
56         return l;
57     }
58
59     // Primer indice con rating > (r + EPS)
60     private int limiteSuperiorRating(double rating) {
61         double objetivo = rating + Movie.EPS;
62         int l = 0;
63         int rr = movies.size();
64         while (l < rr) {
65             int mid = l + (rr - l) / 2;
66             double val = movies.get(mid).getRating();
67             if (val <= objetivo) l = mid + 1;
68             else rr = mid;
69         }
70         return l;
71     }

```

- `public ArrayList<Movie>getMoviesByRatingRange(double lowerRating, double higherRating)`: retorna una lista de películas cuya calificación esta dentro del rango buscado.

Su implementación:

```

1         public ArrayList<Movie> getMoviesByRatingRange(double
2             lowerRating, double higherRating) {
3             ArrayList<Movie> listaRango = new ArrayList<>();
4
5             // Caso 1(lista vacia o el rango es invalido)
6             if (movies.isEmpty() || lowerRating > higherRating) {
7                 return listaRango;
8             }
9
10            // Si se esta ordenado por rating, se prioriza
11            // búsqueda binaria (+eficiente)
12            if ("rating".equals(sortedByAttribute)) {
13                //Comienzo

```

```

12         int lo = limiteInferiorRating(lowerRating);
13         //Final
14         int hi = limiteSuperiorRating(higherRating) - 1;
15
16         for (int i = lo; i <= hi && i < movies.size();
17             i++) {
18             double rat = movies.get(i).getRating();
19             if (rat + Movie.EPS >= lowerRating && rat -
20                 Movie.EPS <= higherRating) {
21                 listaRango.add(movies.get(i));
22             }
23         }
24     } else {
25         // Si no esta ordenado se usa busqueda lineal
26         for (int i = 0; i < movies.size(); i++) {
27             Movie m = movies.get(i);
28             double rat = m.getRating();
29             if (rat + Movie.EPS >= lowerRating && rat -
30                 Movie.EPS <= higherRating) {
31                 listaRango.add(m);
32             }
33         }
34     }
35     return listaRango;
36 }

```

- `public ArrayList<Movie>getMoviesByGenre(String genre):` retorna una lista de películas que pertenezcan exactamente al mismo genero o categoría que el parámetro que se esta buscando.

Su implementación:

```

1     public ArrayList<Movie> getMoviesByGenre(String genre)
2     {
3         ArrayList<Movie> listaGenero = new ArrayList<>();
4
5         //Si "movies" es nulo o esta vacio o el director es
6         //nulo retorna la lista vacia
7         if (movies == null || movies.isEmpty() || genre ==
8             null) {
9             return listaGenero;
10        }
11
12        String generoBuscado = genre.trim();
13
14        if ("genre".equals(sortedByAttribute)) {
15            int first = primerIndiceGenero(generoBuscado);
16            if (first == -1) {
17                return listaGenero;
18            }
19
20            int last = ultimoIndiceGenero(generoBuscado);
21            for (int i = first; i <= last; i++) {

```

---

```

19         listaGenero.add(movies.get(i));
20     }
21     } else {
22         //Si no esta ordenado se usa busqueda lineal
23         for (int i = 0; i < movies.size(); i++) {
24             Movie m = movies.get(i);
25             if (m.getGenre() != null &&
                m.getGenre().equalsIgnoreCase(generoBuscado))
            {
26                 listaGenero.add(m);
27             }
28         }
29     }
30     return listaGenero;
31 }
32
33 private int primerIndiceGenero(String objetivo) {
34     //Comienzo
35     int lo = 0;
36     //Final
37     int hi = movies.size() - 1;
38     //Respuesta primero
39     int respuestaPri = -1;
40
41     //Mientras lo sea menor que hi
42     while (lo <= hi) {
43         //Se define la mitad
44         int mid = lo + (hi - lo) / 2;
45
46         String gen = movies.get(mid).getGenre();
47
48         int comparadorGenero = compareGenero(gen,
            objetivo);
49
50         //Se mueve a la izquierda para encontrar el primero
51         if (comparadorGenero >= 0) {
52             if (comparadorGenero == 0) {
53                 respuestaPri = mid;
54                 hi = mid - 1;
55             }
56         } else {
57             lo = mid + 1;
58         }
59     }
60     return respuestaPri;
61 }
62
63 private int ultimoIndiceGenero(String objetivo) {
64     //Comienzo
65     int lo = 0;
66     //Final
67     int hi = movies.size() - 1;

```



```

68         //Respuesta ultimo
69         int respuestaUlt = -1;
70
71         //Mientras lo sea menor o igual que hi
72         while (lo <= hi) {
73             //Se define la mitad
74             int mid = lo + (hi - lo) / 2;
75
76             String gen = movies.get(mid).getGenre();
77
78             int comparadorGenero = compareGenero(gen,
79                 objetivo);
80
81             //Se mueve a la derecha para encontrar el ultimo
82             if (comparadorGenero <= 0) {
83                 if (comparadorGenero == 0) {
84                     respuestaUlt = mid;
85                     lo = mid + 1;
86                 } else {
87                     hi = mid - 1;
88                 }
89             }
90             return respuestaUlt;
91     }

```

- `public ArrayList<Movie>getMoviesByDirector(String director):` retorna una lista de películas cuyo director cinematográfico sea exactamente igual al que se esta buscando.

Su implementación:

```

1         public ArrayList<Movie> getMoviesByDirector(String
2             director) {
3             ArrayList<Movie> listaDirectores = new ArrayList<>();
4             //Si "movies" es nulo o esta vacio o el director es
5             //nulo retorna la lista vacia
6             if (movies == null || movies.isEmpty() || director ==
7                 null) {
8                 return listaDirectores;
9             }
10
11             String directorBuscado = director.trim();
12
13             if ("director".equals(sortedByAttribute)) {
14
15                 int first = primerIndice(directorBuscado);
16                 if (first == -1) {
17                     return listaDirectores;
18                 }
19
20                 int last = ultimoIndice(directorBuscado);
21                 for (int i = first; i <= last; i++) {

```

```

19         listaDirectores.add(movies.get(i));
20     }
21
22     } else {
23         // Si no esta ordenado se usa busqueda lineal
24         for (int i = 0; i < movies.size(); i++) {
25             Movie m = movies.get(i);
26             if (m.getDirector() != null &&
                m.getDirector().equalsIgnoreCase(directorBuscado))
            {
27                 listaDirectores.add(m);
28             }
29         }
30     }
31     return listaDirectores;
32 }
33
34 // Devuelve el primer ndice
35 private int primerIndice(String objetivo) {
36     //Comienzo
37     int lo = 0;
38     //Final
39     int hi = movies.size() - 1;
40     //Respuesta del primero
41     int respuestaPri = -1;
42
43     //Mientras lo sea menor que hi
44     while (lo <= hi) {
45         //Se define la mitad
46         int mid = lo + (hi - lo) / 2;
47
48         String dir = movies.get(mid).getDirector();
49
50         int comparadorDirector = compareDirector(dir,
                objetivo);
51         //Se mueve a la izquierda para encontrar el primer
            indice
52         if (comparadorDirector >= 0) {
53             if (comparadorDirector == 0) {
54                 respuestaPri = mid;
55                 hi = mid - 1;
56             }
57         }
58         //Se mueve a la derecha para encontrar el primer
            indice
59         else {
60             lo = mid + 1;
61         }
62     }
63     return respuestaPri;
64 }
65

```

```

66 // Devuelve el ultimo ndice
67 private int ultimoIndice(String objetivo) {
68     //Comienzo
69     int lo = 0;
70     //Final
71     int hi = movies.size() - 1;
72     //Respuesta del ultimo
73     int respuestaUlt = -1;
74
75     //Mientras lo sea menor que hi
76     while (lo <= hi) {
77         //Se define la mitad
78         int mid = lo + (hi - lo) / 2;
79
80         String dir = movies.get(mid).getDirector();
81
82         int comparadorDirector = compareDirector(dir,
            objetivo);
83
84         //Se mueve a la derecha para encontrar el ultimo
85         if (comparadorDirector <= 0) {
86             if (comparadorDirector == 0) {
87                 respuestaUlt = mid;
88                 lo = mid + 1;
89             }
90
91         }
92         //Se mueve a la derecha para encontrar el primer
            indice
93         else {
94             hi = mid - 1;
95         }
96     }
97     return respuestaUlt;
98 }
99
100 // Comparador de director
101 //Metodo que ve si una cadena es menor, mayor o igual que
    otra
102 private int compareDirector(String a, String b) {
103     //Si son iguales retorna 0
104     if (a == null && b == null) {
105         return 0;
106     }
107     //La primera cadena es menor que la segunda
108     if (a == null) {
109         return -1;
110     }
111     //La primera cadena es mayor que la segunda
112     if (b == null) {
113         return 1;
114     }

```

```

115         return a.compareToIgnoreCase(b);
116     }

```

- `public ArrayList<Movie>getMoviesByYear(int year)`: método que retorna una lista de películas estrenadas en el año del parámetro que se entrega.

Su implementación:

```

1     public ArrayList<Movie> getMoviesByYear(int year) {
2         ArrayList<Movie> listaYear = new ArrayList<>();
3         //Si "movies" es nulo o esta vacio o el director es
4         //nulo retorna la lista vacia
5         if (movies == null || movies.isEmpty()) {
6             return listaYear;
7         }
8         if ("year".equals(sortedByAttribute)) {
9
10            int first = primerIndiceYear(year);
11            if (first == -1) {
12                return listaYear;
13            }
14
15            int last = ultimoIndiceYear(year);
16            for (int i = first; i <= last; i++) {
17                listaYear.add(movies.get(i));
18            }
19        } else {
20            // Si no esta ordenado se usa busqueda lineal
21            for (int i = 0; i < movies.size(); i++) {
22                Movie m = movies.get(i);
23                if (m.getReleaseYear() == year) {
24                    listaYear.add(m);
25                }
26            }
27        }
28        return listaYear;
29    }
30
31    // Devuelve el primer ndice
32    private int primerIndiceYear(int objetivo) {
33        //Comienzo
34        int lo = 0;
35        //Final
36        int hi = movies.size() - 1;
37        //Respuesta primero
38        int respuestaPri = -1;
39
40        //Mientras lo sea menor o igual a hi
41        while (lo <= hi) {
42            //Se define la mitad
43            int mid = lo + (hi - lo) / 2;
44

```

```

45         int ye = movies.get(mid).getReleaseYear();
46
47         int comparadorYear = compareYear(ye, objetivo);
48         if (comparadorYear >= 0) { // explorar a
49             la izquierda para hallar el primero
50             if (comparadorYear == 0) {
51                 respuestaPri = mid;
52                 hi = mid - 1;
53             }
54         } else {
55             lo = mid + 1;
56         }
57     }
58     return respuestaPri;
59 }
60
61 // Devuelve el ultimo indice
62 private int ultimoIndiceYear(int objetivo) {
63     //Comienzo
64     int lo = 0;
65     //Final
66     int hi = movies.size() - 1;
67     //Respuesta ultimo
68     int respuestaUlt = -1;
69
70     //Mientras lo sea menor o igual a hi
71     while (lo <= hi) {
72         //Se define la mitad
73         int mid = lo + (hi - lo) / 2;
74
75         int ye = movies.get(mid).getReleaseYear();
76
77         int comparadorYear = compareYear(ye, objetivo);
78         if (comparadorYear <= 0) { // explorar a
79             la derecha para hallar el ltimo
80             if (comparadorYear == 0) {
81                 respuestaUlt = mid;
82                 lo = mid + 1;
83             }
84         } else {
85             hi = mid - 1;
86         }
87     }
88     return respuestaUlt;
89 }
90
91 //Comparador para el a o
92 private int compareYear(int a, int b) {
93     return Integer.compare(a, b);
94 }

```

---

En resumen el catalogo puede localizar películas según los criterios de: rating, rango del rating, genero, director y año de lanzamiento de la película.

Se aplicaron dos formas de búsqueda dentro de los métodos, estos son:

- Búsqueda lineal  
Recorre secuencialmente la lista, esta búsqueda se utiliza cuando el catálogo no está ordenado por el atributo de búsqueda. Su complejidad es de  $O(n^2)$ .
- Búsqueda binaria  
Se utiliza cuando el catálogo se encuentra ordenado previamente por el atributo seleccionado. Permite encontrar elementos de forma mucho más rápida. Su complejidad es de  $O(n \log n)$

## 2.6. clase *CsvIMDBLoader*

Esta clase esta encargada de leer y procesar el dataset `imdb_top_1000.csv`.

Para poder dividir las celdas correctamente, ya que estas pueden presentar comas dentro de las comillas, se uso la expresión `Pattern SPLIT`.

Cada una de las filas se transforma en un objeto de tipo `Movie` y se añade a un `ArrayList<Movie>`, el cual despúes es entregado a la clase `MovieCatalog`.

## 2.7. Clase Main

Esta clase se encarga de coordinar los procesos de carga, ordenamiento, búsqueda y medición de tiempo. Se implementaron ciclos para probar subconjuntos de datos de tamaño incremental (100, 200, 300, 400, 500, 600, 700, 800, 900 y 1000 películas), y se registran los tiempos de ejecución de cada algoritmo.

# 3. Experimentación

Todas las pruebas experimentales se realizaron en lenguaje de *Java*, ejecutado en el entorno de desarrollo *IntelliJ IDEA*. Para la medición del tiempo de ejecución se utilizo un método que lo mide en nanosegundos llamado `System.nanoTime()`.

## 3.1. Experimento 1

El objetivo de este primer experimento es de comparar la eficiencia de los algoritmos de ordenamiento implementados en el código, sobre subconjuntos crecientes del dataset *IMDb Top 1000 Movies and TV Shows*, los algoritmos fueron los siguientes:

- InsertionSort
- SelectionSort

- 
- MergeSort
  - QuickSort
  - RadixSort

### Metodología

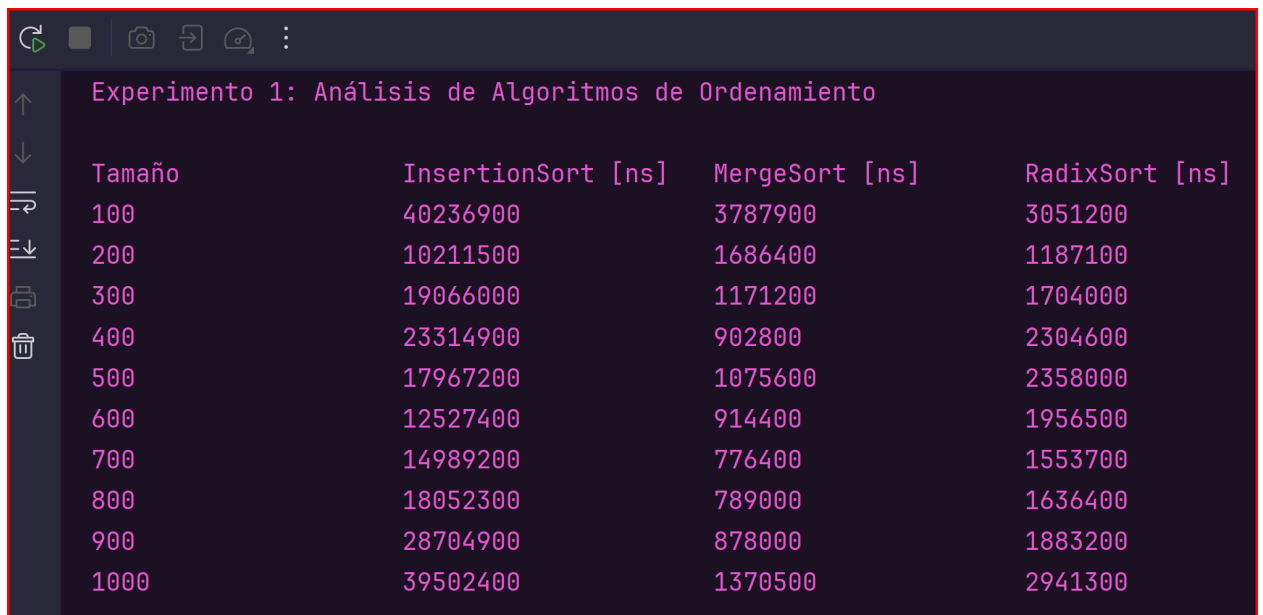
1. Se cargó el dataset completo en un `ArrayList<Movie>`.
2. Para cada tamaño de muestra  $n$  el cual tiene un subconjunto creciente (100, 200, ..., 1000), se crearon cinco catálogos.
3. Cada catálogo fue ordenado por el atributo **rating** utilizando algoritmos diferentes.
4. El tiempo de ejecución se midió en nanosegundos (ns), utilizando el método `System.nanoTime()`.
5. Los resultados se promediaron tras varias ejecuciones para evitar variaciones significativas en el sistema.

En el siguiente cuadro 1 se puede apreciar el tiempo de ejecución promedio por algoritmo:

Cuadro 1: Comparación de tiempos de ejecución de algoritmos de ordenamiento

Tamaño (n)	InsertionSort (ns)	MergeSort (ns)	RadixSort (ns)
100	40236900	3787900	3051200
200	10211500	1686400	1187100
300	19066000	1686400	1704000
400	23314900	902800	2304600
500	17967200	1075600	2358000
600	12527400	914400	1956500
700	14989200	776400	1553700
800	18052300	789000	1636400
900	28704900	878000	1883200
1000	39502400	1370500	2941300

En la siguiente figura 1 se puede ver lo mostrado por la consola al ejecutar el código:



Tamaño	InsertionSort [ns]	MergeSort [ns]	RadixSort [ns]
100	40236900	3787900	3051200
200	10211500	1686400	1187100
300	19066000	1171200	1704000
400	23314900	902800	2304600
500	17967200	1075600	2358000
600	12527400	914400	1956500
700	14989200	776400	1553700
800	18052300	789000	1636400
900	28704900	878000	1883200
1000	39502400	1370500	2941300

Figura 1: Experimento 1A: Análisis de Algoritmos de Ordenamiento

En el siguiente cuadro 2 se puede apreciar el tiempo de ejecución promedio por algoritmo:

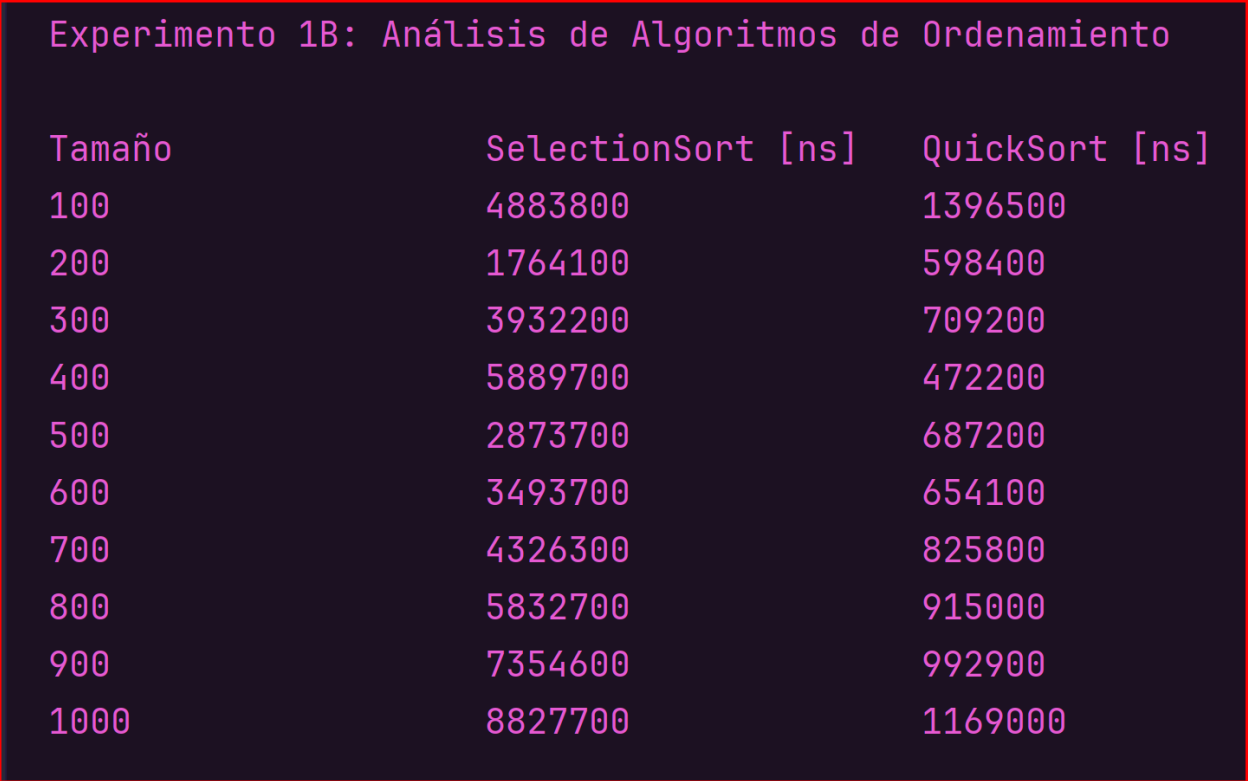
Cuadro 2: Comparación de tiempos de ejecución de algoritmos de ordenamiento

Tamaño (n)	SelectionSort (ns)	QuickSort (ns)
100	4883800	1396500
200	1764100	598400
300	3932200	709200
400	5889700	472200
500	2873700	687200
600	3493700	654100
700	4326300	825800
800	5832700	915000
900	7354600	992900
1000	8827700	1169000



---

En la siguiente figura 2 se puede ver lo mostrado por la consola al ejecutar el código:



Tamaño	SelectionSort [ns]	QuickSort [ns]
100	4883800	1396500
200	1764100	598400
300	3932200	709200
400	5889700	472200
500	2873700	687200
600	3493700	654100
700	4326300	825800
800	5832700	915000
900	7354600	992900
1000	8827700	1169000

Figura 2: Experimento 1B: Análisis de Algoritmos de Ordenamiento

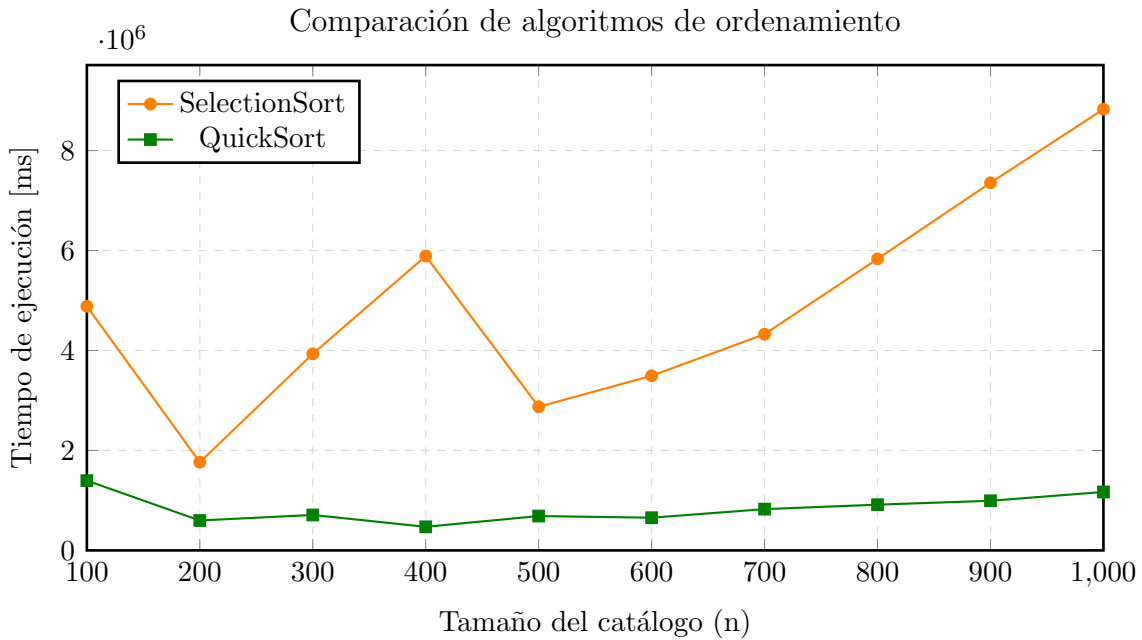


Figura 3: Comparación del tiempo de ejecución de SelectionSort y QuickSort según el tamaño del catálogo.

### Análisis

Con este primer experimento se evidencia que para catálogos pequeños los algoritmos cuadráticos son funcionales y fáciles de implementar, mientras que para volúmenes grandes de datos conviene emplear algoritmos logarítmicos o no comparativos.

*QuickSort* y *RadixSort* se consolidaron como los más eficientes en este laboratorio, seguidos de cerca por *MergeSort*, mientras que *InsertionSort* y *SelectionSort* fueron útiles principalmente como referencia teórica y comparativa.

## 3.2. Experimento 2

El objetivo de este segundo experimento es de comparar la eficiencia en las búsquedas lineales y las búsquedas binarias, al buscar películas según el nombre del director en subconjuntos de distintos tamaños.

### Metodología

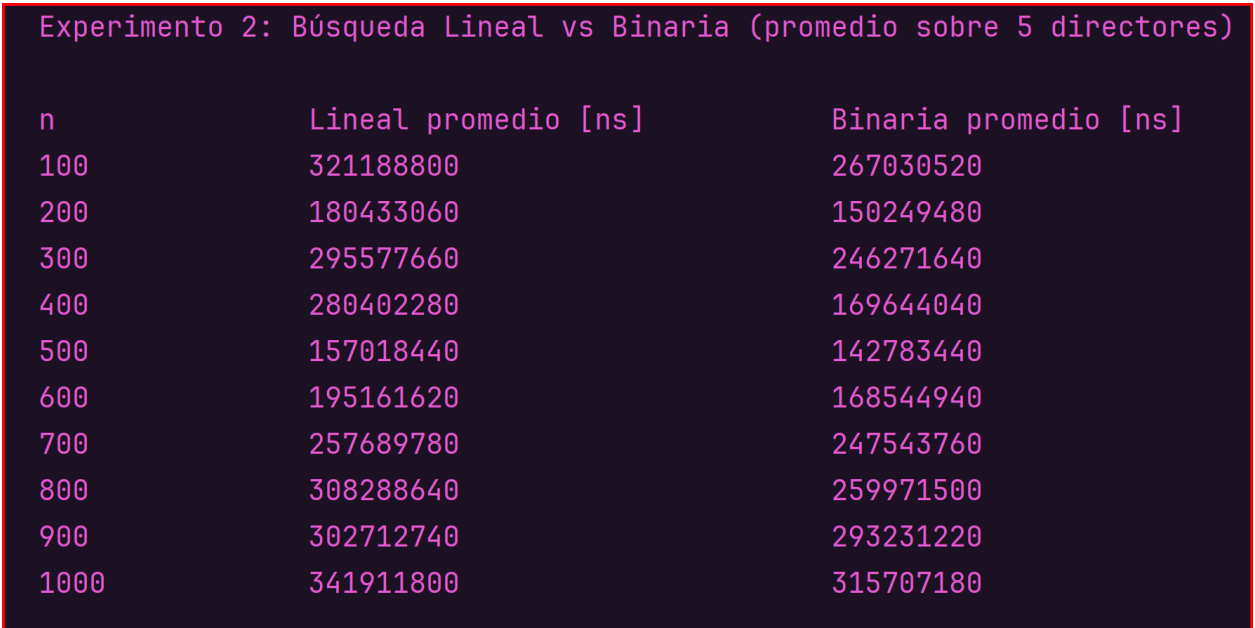
1. Se seleccionaron 5 directores del dataset.
2. Para cada uno de los subconjuntos se midieron los tiempos de búsqueda de cada director. Si este se encontraba desordenado, se utiliza búsqueda lineal y si el catálogo está ordenado se utiliza búsqueda binaria.
3. Se registraron los tiempos promedio de ambas estrategias.

En el siguiente cuadro 3 se muestran los tiempos obtenidos:

Cuadro 3: Comparación de tiempos de búsqueda lineal y binaria

Tamaño de catalogo (n)	Búsqueda lineal (ms)	Búsqueda binaria (ms)
100	321188800	267030520
200	180433060	150249480
300	295577660	246271640
400	280402280	169644040
500	157018440	142783440
600	195161620	168544940
700	257689780	247543760
800	308288640	259971500
900	302712740	293231220
1000	341911800	315707180

En la siguiente imagen se puede visualizar el resultado por consola:



```
Experimento 2: Búsqueda Lineal vs Binaria (promedio sobre 5 directores)
```

n	Lineal promedio [ns]	Binaria promedio [ns]
100	321188800	267030520
200	180433060	150249480
300	295577660	246271640
400	280402280	169644040
500	157018440	142783440
600	195161620	168544940
700	257689780	247543760
800	308288640	259971500
900	302712740	293231220
1000	341911800	315707180

Figura 4: Comparación de tiempo de búsqueda lineal y binaria.

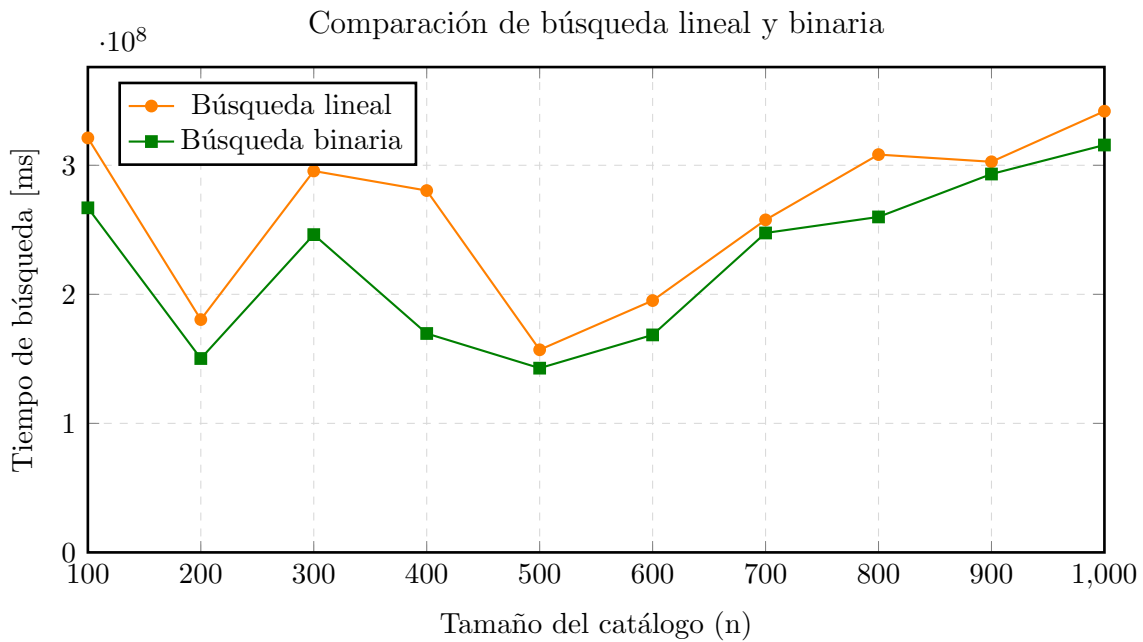


Figura 5: Comparación de tiempos de ejecución de búsqueda lineal y binaria según el tamaño del catálogo.

### Análisis

Los resultados obtenidos demuestran una clara diferencia en el rendimiento entre ambas búsquedas. Mientras la búsqueda lineal presenta un crecimiento proporcional al tamaño del catálogo, con una complejidad de  $O(n)$ , la búsqueda binaria mantiene tiempos casi constantes o con incrementos mínimos al aumentar  $n$ , evidenciando su complejidad  $O(\log n)$ .

En catálogos pequeños la diferencia no es tan perceptible, pero a medida que el número de películas aumenta, la búsqueda binaria se vuelve considerablemente más eficiente.

---

## 4. Conclusión

A partir de los resultados obtenidos, se evidencio una diferencia entre los algoritmos de ordenamiento y búsqueda. En los algoritmos de ordenamiento, *QuickSort* presentó tiempos de ejecución mucho menores que *SelectionSort*, especialmente al aumentar el tamaño de los datos. Esto confirma su mayor eficiencia, ya que *QuickSort* tiene una complejidad promedio de  $O(n \log n)$ , mientras que *SelectionSort* alcanza  $O(n^2)$ .

En cuanto a los algoritmos de búsqueda se evidencio que la búsqueda binaria fue mas rápida que la búsqueda lineal, manteniendo menor tiempo en casi todos los casos. Esto se debe a que la búsqueda binaria reduce considerablemente el número de comparaciones al trabajar sobre datos ordenados, con una complejidad  $O(\log n)$  frente a la  $O(n)$  de la búsqueda lineal.