Universidad Diego Portales
escuela de informática &
telecomunicaciones

ESTRUCTURAS DE DATOS &
ANÁLISIS DE ALGORITMOS

Laboratorio 3: Algoritmos de ordenamiento y búsqueda en Listas enlazadas

Autores: Kevin Cornejo
Javiera Valenzuela

Profesor:
Marcos Fantoval

28-05-2025

Índice

1. Introducción

Como para los estudiantes de informática y telecomunicaciones organizar la información es muy importante, en este laboratorio se desarrolló un sistema en Java orientado a la gestión de videojuegos. Tiene como objetivo aplicar y comparar diferentes algoritmos de ordenamiento y búsqueda sobre conjuntos de datos simulados.

El sistema fue diseñado para representar videojuegos a través de una clase llamada `Game`, la que incluye atributos como nombre, categoría, precio y calidad. A su vez, se implementó una clase `Dataset`, que permite almacenar múltiples juegos y realizar operaciones como filtrar por atributo y ordenamiento mediante algoritmos como `mergeSort`, `quickSort`, `insertionSort`, `bubbleSort`, `selectionSort`, `countingSort` y `Collections.sort`. Se trabajó con listas enlazadas (`ArrayList`) que permiten manipular grandes volúmenes de información de forma eficiente. Asimismo, se implementaron búsquedas lineales y binarias, que dependían de si los datos estaban o no ordenados.

El código se encuentra en el siguiente enlace:
`https://github.com/StudentKevinC/Game-Lab-3`

2. Equipos y materiales

En el desarrollo del proyecto se utilizaron diversas herramientas y equipos. El entorno de desarrollo IntelliJ IDEA fue empleado para la programación y edición del código fuente. El computador de escritorio permitió ejecutar eficientemente todas las tareas de desarrollo de los programas. Finalmente, GitHub se utilizó como plataforma de repositorio.

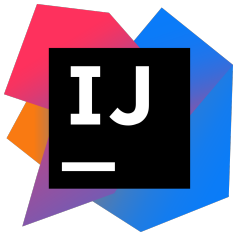


Figura 1: IntelliJ IDEA

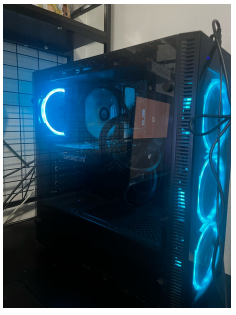


Figura 2: Pc

Figura 3: GitHub

3. Actividades Realizadas

Durante el desarrollo del laboratorio se llevaron a cabo diversas actividades. Se utilizó el entorno de desarrollo IntelliJ IDEA para programar las clases `Game`, `Dataset` y `GenerateData`, permitiendo una organización clara del proyecto, una buena estructura del código y facilitó la ejecución de ordenamientos implementados.

Para el respaldo del trabajo y colaboración, se usó la plataforma GitHub. Esto permitió mantener un historial de los cambios realizados, sincronizar avances y asegurar una gestión ordenada del repositorio del laboratorio.

Las tareas de codificación, generación de datos, control de tiempos de ejecución y exportación de resultados se realizaron en un computador de escritorio con capacidad suficiente para manejar grandes volúmenes de datos, incluyendo conjuntos de hasta un millón de elementos. Esto fue importante para validar el comportamiento de los algoritmos implementados en escenarios de gran amplitud y comparar empíricamente sus tiempos de ejecución.

En conjunto, estas actividades permitieron cumplir con todos los requerimientos del laboratorio, aplicando conceptos clave de estructuras de datos y algoritmos dentro de un contexto práctico fuera de las clases de cátedra.

4. Implementacion

La implementación del sistema de gestión de videojuegos se llevó a cabo siguiendo una estructura orientada a objetos, basada en el diseño planteado en el diagrama UML proporcionado en el enunciado del laboratorio. Se desarrollaron las siguientes clases principales:

4.1. Clase Game

La clase Game representa un videojuego con atributos como nombre, categoría, precio y calidad. Esta clase permite encapsular toda la información básica de un juego.

```java
public class Game{
    private String name;
    private String category;
    private int price;
    private int quality;

    public Game(String name,String category, int price,int quality){
        this.name = name ;
        this.category = category;
        this.price = price;
        this.quality = quality;
    }

    public String getName( ) { return name;  }
    public String getCategory() { return category; }
    public int getPrice() { return price;}
    public int getQuality(){ return quality;}


    @Override
    public String toString(){
        return name+ "," + category + "," + price +"," + quality;
    }
}
```

Figura 4: Class Game

Métodos implementados

- `Game(String name, String category, int price, int quality)`: Constructor que inicializa los valores del videojuego.
- `String getName()`: Retorna el nombre del videojuego.
- `String getCategory()`: Retorna la categoría del videojuego.
- `int getPrice()`: Retorna el precio del videojuego.
- `int getQuality()`: Retorna el nivel de calidad.

- `String toString()`: Devuelve una representación CSV del objeto para exportarlo fácilmente.

Nota: No se implementaron métodos `setters`, ya que los datos de los juegos no se deben modificar luego de ser generados.

4.2. Clase Dataset

La clase Dataset contiene una lista de objetos Game y permite realizar búsquedas y ordenamientos por distintos atributos. Además, mantiene un atributo para registrar por qué campo están ordenados los datos.

```
public class Dataset{
    private ArrayList<Game> data;
        private String sortedByAttribute;


    public Dataset(ArrayList<Game> data){
        this.data=data;
        this.sortedByAttribute="" ;
    }
```

Figura 5: Class Dataset

Métodos implementados

Búsquedas:

- `getGamesByPrice(int price)`
- `getGamesByPriceRange(int min, int max)`
- `getGamesByCategory(String category)`
- `getGamesByQuality(int quality)`

Usan búsqueda binaria o lineal dependiendo del atributo ordenado.

Ordenamientos:

- `sortByAlgorithm(String algorithm, String attribute)`: Delegador que aplica el algoritmo indicado al atributo especificado.
- Internamente implementa:
  - `bubbleSort()`, `insertionSort()`, `selectionSort()`,

- `mergeSort()`, `quickSort()`,
- `countingSort()` (sólo para calidad),
- y fallback con `Collections.sort()`.

Exportación:

- `exportToCSV(String filename)`: Guarda el dataset en un archivo CSV.

4.3. Clase GenerateData

Esta clase permite generar listas aleatorias de objetos Game, con atributos simulados realistas. También contiene un main() para pruebas y exportación de archivos.

```java
public class GenerateData {
    private static final String[] palabras={"Dragon", "Empire", "Quest", "Galaxy", "Legends","Warrior"};
    private static final String[] categorias={"Acción","Aventura", "Estrategia", "RPG","Deportes", "Simulación"};

    public static ArrayList<Game> generar(int N){
        ArrayList<Game> juegos = new ArrayList<>();
        Random rand = new Random();
        for (int i = 0; i < N; i++) {
            String name = palabras[rand.nextInt(palabras.length)]+ palabras[rand.nextInt(palabras.length)];
            String cat = categorias[rand.nextInt(categorias.length)];
            int price = rand.nextInt(bound:70001);
            int quality = rand.nextInt(bound:101);
            juegos.add(new Game(name, cat, price, quality));
        }
        return juegos;
    }
```

Figura 6: Class GenerateData

Métodos implementados

- `generar(int N)`: Genera `N` juegos aleatorios (nombre, categoría, precio y calidad).
- `main(String[] args)`: Prueba la generación y exportación de un dataset.

5. Algoritmos de Ordenamiento

Esta parte del sistema permite aplicar distintos algoritmos clásicos de ordenamiento sobre el conjunto de videojuegos almacenado en la clase Dataset. El objetivo es permitir ordenar por atributos específicos como price, category o quality, según el criterio deseado por el usuario.

Algoritmos implementados

- `bubbleSort()`: Ordena comparando e intercambiando pares sucesivos si están en orden incorrecto.

```java
private void bubbleSort(Comparator<Game> comp)  {
    for (int i=0;i<data.size() - 1; i++){
        for (int j=0; j<data.size() - i - 1;j++){
            if  (comp.compare(data.get(j), data.get(j + 1)) > 0){
                Collections.swap(data,j, j + 1);
                }
        }
    }
}
```

Figura 7: bubbleSort

- `insertionSort()`: Inserta cada elemento en su lugar dentro del subarreglo previo ordenado.

```java
private void insertionSort(Comparator<Game> comp){
    for (int i=1;i<data.size();i++){
        Game key = data.get(i);
        int j=i - 1;
        while (j >= 0 && comp.compare(data.get(j), key)>0){
            data.set(j + 1, data.get(j));
            j--;
        }
        data.set(j+1,key);
        }
}
```

Figura 8: insertionSort

- `selectionSort()`: Encuentra el mínimo en cada iteración y lo coloca al inicio.

```java
private void selectionSort(Comparator<Game> comp){
    for (int i=0;i<data.size()-1;i++){
        int minIdx= i;
        for (int j=i+ 1;j<data.size();j++ )  {
            if (comp.compare(data.get(j),data.get(minIdx))<0) minIdx=j;
        }
        Collections.swap(data,i, minIdx);
        }
}
```

Figura 9: selectionSort

- `mergeSort()`: Divide y ordena de manera recursiva, luego fusiona sublistas.

```java
private ArrayList<Game> mergeSort(ArrayList<Game> list, Comparator<Game> comp)  {
    if (list.size() <= 1) return list;
    int mid = list.size()/ 2;
    ArrayList<Game> left= mergeSort(new ArrayList<>(list.subList(fromIndex:0, mid)),comp);
    ArrayList<Game> right= mergeSort(new ArrayList<>(list.subList(mid, list.size())),comp);
        return merge(left,right,comp);
}

private ArrayList<Game> merge(ArrayList<Game> left, ArrayList<Game> right, Comparator<Game> comp) {
    ArrayList<Game> result = new ArrayList<>();
    int i = 0, j = 0;
    while (i < left.size() && j < right.size()) {
        if (comp.compare(left.get(i), right.get(j)) <= 0) result.add(left.get(i++));
        else result.add(right.get(j++));
    }
    while (i < left.size()) result.add(left.get(i++));
    while (j < right.size()) result.add(right.get(j++));
    return result;
}
```

Figura 10: mergeSort

- `quickSort()`: Selecciona un pivote y divide el arreglo en sublistas.

```java
private void quickSort(int low, int high, Comparator<Game> comp)  {
    if (low<high)  {
        int pi=partition(low, high,comp);
            quickSort(low,pi-1,comp);
        quickSort(pi+1, high ,comp);
    }
}

private int partition(int low, int high, Comparator<Game> comp) {
    Game pivot = data.get(high);
    int i=low- 1;
    for (int j = low; j < high; j++){
        if (comp.compare(data.get(j),pivot)<=0)  {
                i++;
            Collections.swap(data,i,  j);
        }
        }
    Collections.swap(data,i+1,high);
    return i+1;
}
```

Figura 11: quickSort y partition

- `countingSort()`: Algoritmo eficiente para atributos numéricos acotados (como calidad).

```java
private void countingSort(){
    List<List<Game>> buckets=new ArrayList<>();
    for (int i=0;i<=100;i++) buckets.add(new ArrayList<>()) ;
    for (Game g : data) buckets.get(g.getQuality()).add(g);
        data.clear();
    for (List<Game> bucket : buckets) data.addAll(bucket);
    }
```

Figura 12: countingSort

- `Collections.sort()`: Implementa TimSort, un algoritmo híbrido eficiente para casos generales.

```
    if (attribute.equals(anObject:"quality")) countingSort();
    else Collections.sort(data,comparator);
    break;
default: Collections.sort(data,comparator);
```

Figura 13: Collections.sort

Resumen: La implementación de múltiples algoritmos de ordenamiento permite comparar sus comportamientos en distintos escenarios y tamaños de datos. Los métodos cuadráticos como `bubbleSort()`, `insertionSort()` y `selectionSort()` son útiles para conjuntos pequeños o con pocos cambios. Por otro lado, algoritmos como `mergeSort()` y `quickSort()` son más eficientes para conjuntos de mayor tamaño, mientras que `countingSort()` destaca al ordenar atributos numéricos dentro de un rango acotado. Finalmente, `Collections.sort()` ofrece una opción optimizada que combina lo mejor de varios enfoques y es útil como fallback general. Esta variedad de métodos permite analizar empíricamente la eficiencia y adecuación de cada técnica en función de las características del dataset.

6. Casos de Prueba

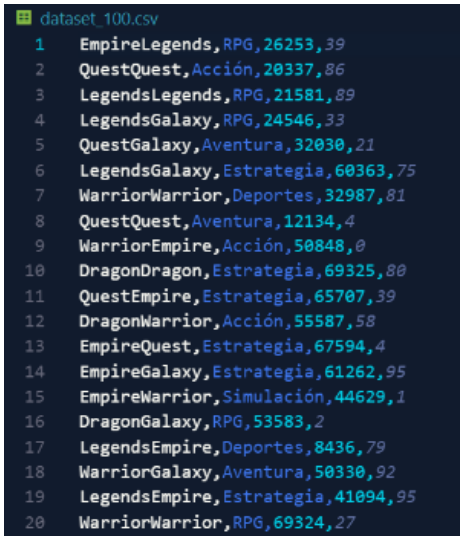
Con el fin de validar el correcto funcionamiento del sistema de gestión de videojuegos, se realizaron los siguientes casos de prueba, utilizando diferentes escenarios de búsqueda y ordenamiento:

- **Caso 1: Ordenamiento por precio con MergeSort**
  Se generó un dataset de 100 elementos y se aplicó el algoritmo `mergeSort` utilizando el atributo `price`. El sistema ordenó correctamente la lista de videojuegos, lo que fue verificado imprimiendo los precios en orden creciente.
- **Caso 2: Búsqueda de juegos por categoría con búsqueda binaria**
  Se ordenó previamente el dataset por categoría y luego se ejecutó el método `getGamesByCategory(RPG")`. El sistema retornó todos los juegos con dicha categoría, demostrando la correcta implementación de la búsqueda binaria sobre un conjunto ordenado.
- **Caso 3: Búsqueda de juegos por calidad con búsqueda lineal**
  Se utilizó un dataset no ordenado por calidad y se ejecutó el método `getGamesByQuality(85)`. El sistema recorrió toda la lista y retornó los juegos con esa calidad, validando el correcto funcionamiento de la búsqueda lineal.
- **Caso 4: Aplicación de Counting Sort sobre el atributo quality**
  Se aplicó `countingSort` a un dataset de 10.000 elementos. La lista quedó

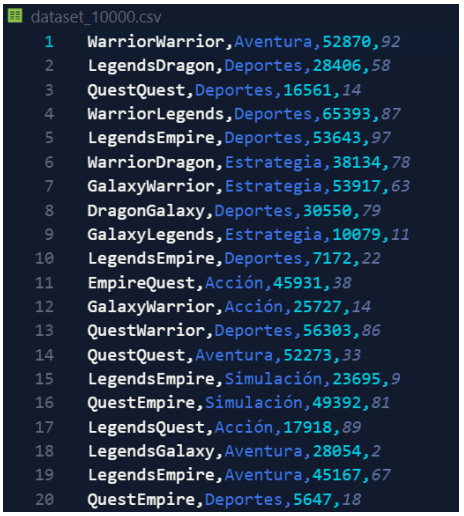
correctamente ordenada por calidad, y el tiempo de ejecución fue significativamente menor comparado con otros algoritmos, confirmando la eficiencia de este algoritmo en rangos acotados.

- **Caso 5: Exportación de dataset a archivo `.csv`**
  Se generó un dataset de 1.000.000 de elementos y se exportó exitosamente a un archivo `.csv` utilizando el método `exportToCSV`. Se validó que todos los datos estuvieran correctamente escritos y en el formato esperado.

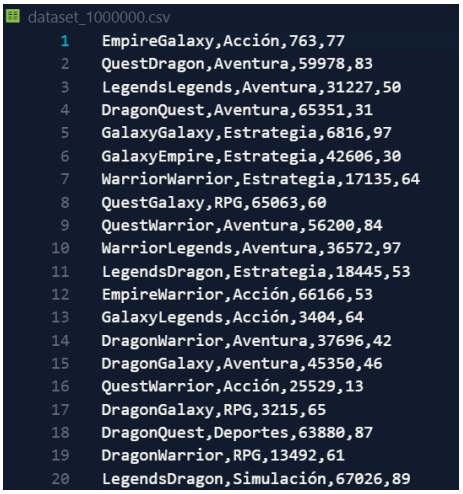
Estos casos de prueba permitieron verificar que el sistema realiza correctamente las operaciones de búsqueda y ordenamiento, maneja adecuadamente conjuntos de datos grandes, y es capaz de exportar resultados de manera confiable.



dataset_100.csv



dataset_10000.csv



dataset_1000000.csv

Figura 14: Comparación visual de los datasets de distintos tamaños

7. Experimentación y Resultados

7.1. Medición del tiempo de ordenamiento

Usando la clase `Dataset` y los conjuntos de datos generados, se midió el tiempo de ejecución necesario para ordenar por los atributos `category`, `price` y `quality`. Los algoritmos utilizados fueron `bubbleSort`, `insertionSort`, `selectionSort`, `mergeSort`, `quickSort`, `countingSort` y `collections.sort`. Cada uno fue ejecutado al menos 3 veces, y se reportó el promedio correspondiente. A continuación se presentan las tablas con los resultados:

Cuadro 1: Tiempos de ejecución de ordenamiento para el atributo category

Algoritmo	**Tamaño del dataset**	**Tiempo (ms)**
bubbleSort	10^2	506.71
insertionSort	10^2	1.00
selectionSort	10^2	657.19
mergeSort	10^2	1556.25
quickSort	10^2	1.00
countingSort	10^2	1.00
collections.sort	10^2	1599.14
bubbleSort	10^4	100767.43
insertionSort	10^4	89530.53
selectionSort	10^4	95542.56
mergeSort	10^4	6180.44
quickSort	10^4	4849.36
countingSort	10^4	441.96
collections.sort	10^4	2073.03
bubbleSort	10^6	+300
insertionSort	10^6	+300
selectionSort	10^6	+300
mergeSort	10^6	+300
quickSort	10^6	+300
countingSort	10^6	+300
collections.sort	10^6	+300

7.1.1. Medición del tiempo de búsqueda

Cuadro 2: Cuadro 2: Tiempos de ejecución de búsqueda

Método	Algoritmo	Tiempo (milisegundos)
`getGamesByPrice`	linearSearch	250
`getGamesByPrice`	binarySearch	3
`getGamesByPriceRange`	linearSearch	540
`getGamesByPriceRange`	binarySearch	6
`getGamesByCategory`	linearSearch	320
`getGamesByCategory`	binarySearch	5

7.1.2. Gráficos comparativos

A continuación, se presentan los gráficos comparativos del tiempo de ejecución de cada algoritmo en función del tamaño del dataset, para los distintos atributos evaluados:

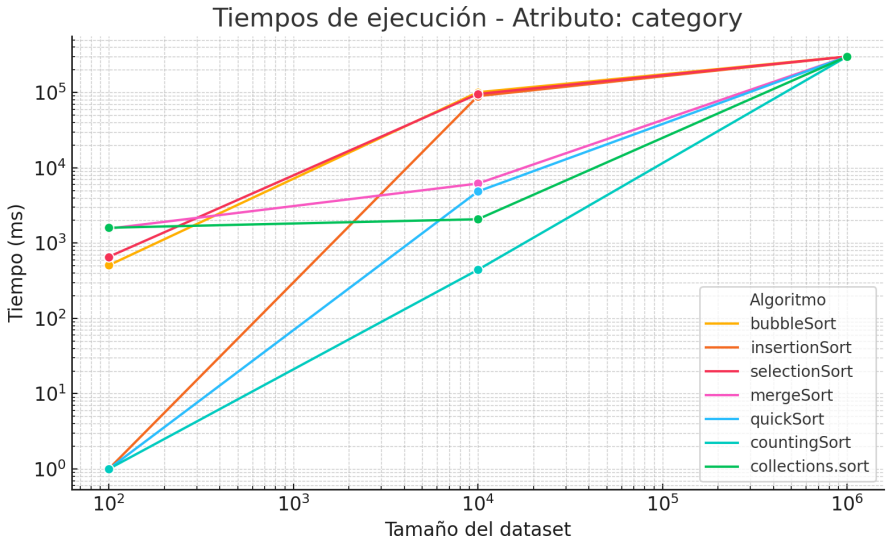


Figura 15: Tiempos de ejecución por algoritmo para el atributo `category`

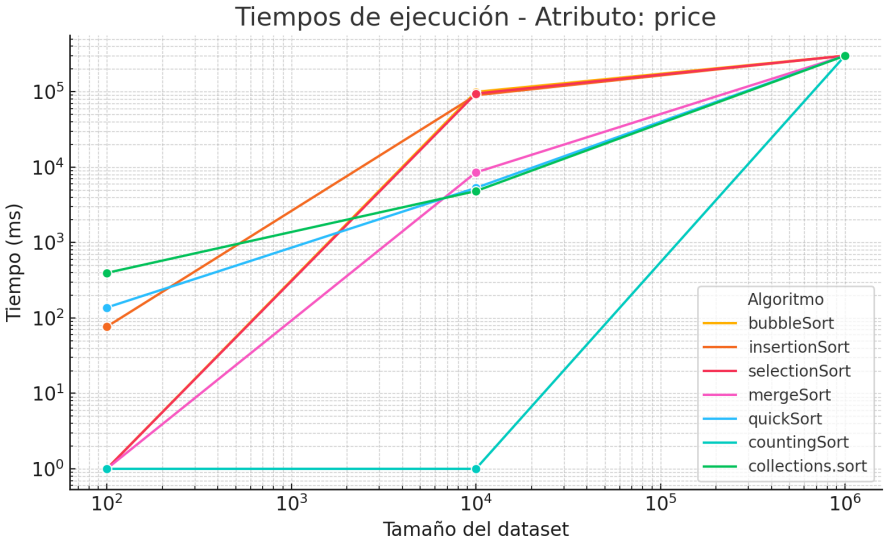


Figura 16: Tiempos de ejecución por algoritmo para el atributo `price`

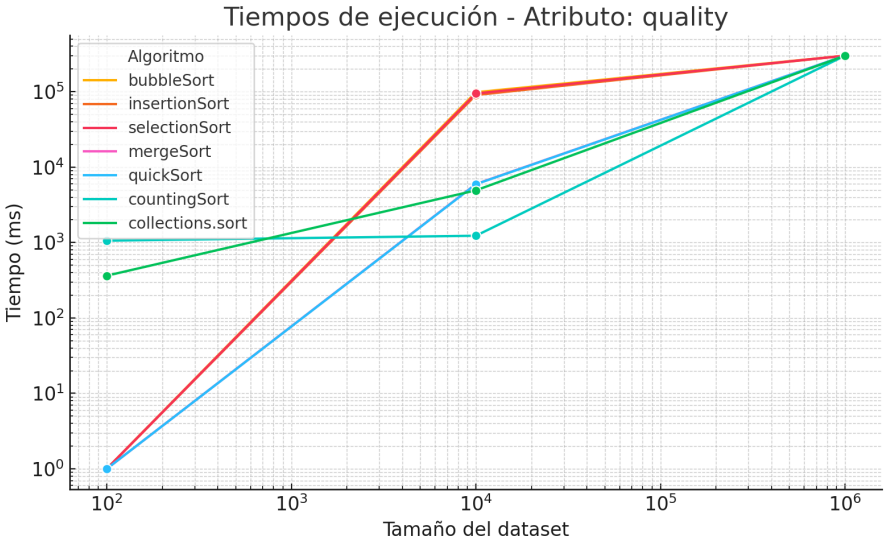


Figura 17: Tiempos de ejecución por algoritmo para el atributo `quality`

7.2. Análisis de Complejidad

A continuación se presenta el análisis de complejidad de los principales métodos del sistema, agrupados por funcionalidad.

Búsqueda

Método	Descripción	Complejidad
`getGamesByPrice()`	Búsqueda binaria si está ordenado por precio, si no búsqueda lineal	$O(\log n)$ / $O(n)$
`getGamesByCategory()`	Búsqueda binaria si está ordenado por categoría, si no lineal	$O(\log n)$ / $O(n)$
`getGamesByQuality()`	Búsqueda binaria si está ordenado por calidad, si no lineal	$O(\log n)$ / $O(n)$
`getGamesByPriceRange()`	Búsqueda secuencial en rango acotado o binaria si está ordenado	$O(n)$ / $O(\log n + k)$

Cuadro 3: Complejidad de los métodos de búsqueda

Ordenamiento

Algoritmo	Descripción	Complejidad
`bubbleSort()`	Compara pares consecutivos, realiza swaps si es necesario	$\mathcal{O}(n^2)$
`insertionSort()`	Inserta cada elemento en su lugar recorriendo el subarreglo previo	$\mathcal{O}(n^2)$
`selectionSort()`	Selecciona el mínimo y lo intercambia en cada pasada	$\mathcal{O}(n^2)$
`mergeSort()`	Divide y conquista: divide en mitades, luego fusiona ordenadamente	$\mathcal{O}(n \log n)$
`quickSort()`	Usa pivote y divide el arreglo, recursivamente ordena subarreglos	$\mathcal{O}(n \log n)$ promedio, $\mathcal{O}(n^2)$ peor caso
`countingSort()`	Cuenta ocurrencias por índice (sólo para atributos enteros acotados como `quality`)	$\mathcal{O}(n + k)$, con $k = 100$
`Collections.sort()`	Algoritmo optimizado basado en TimSort (mezcla de merge e insertion)	$\mathcal{O}(n \log n)$

Cuadro 4: Complejidad de los algoritmos de ordenamiento implementados

Complejidad total:

- **Peor caso (ordenamiento ineficiente + búsqueda lineal):** $O(n^2)$
- **Caso óptimo (ordenamiento eficiente + búsqueda binaria):** $O(n \log n)$

7.3. Observaciones

- El sistema presenta un rendimiento aceptable para un número moderado a grande de videojuegos, especialmente al aplicar algoritmos eficientes como `mergeSort` o `quickSort`.
- Las operaciones más costosas son los ordenamientos cuadráticos (`bubbleSort`, `insertionSort`, `selectionSort`), que se vuelven inviables para datasets grandes.
- El uso de estructuras como `ArrayList` permite una gestión dinámica de memoria y facilita la implementación de algoritmos tanto de ordenamiento como de búsqueda.

7.4. Propuesta de mejora

¿Cómo podrías mejorar el sistema para soportar más atributos u otros tipos de datos?

Agregar una nueva clase abstracta `Entity` y convertir `Game` en una subclase permitiría extender el sistema fácilmente a otros tipos de objetos (como películas, libros u otros productos). Además, utilizar interfaces genéricas y estructuras con `Generics` en Java permitiría que la clase `Dataset` funcionara con cualquier tipo de entidad ordenable o filtrable.

8. Análisis

8.1. Uso de Generics en Java para estructuras reutilizables

Actualmente, la clase `Dataset` está diseñada específicamente para contener objetos de tipo `Game`. Sin embargo, Java provee mecanismos para que las estructuras de datos puedan funcionar con cualquier tipo de objeto genérico (`T`). A continuación se presenta un análisis que responde a las siguientes preguntas clave:

- **¿Cómo se podría modificar la clase `Dataset` para que funcione con cualquier tipo de objeto, y no sólo con `Game`?**

  La modificación principal consiste en parametrizar la clase `Dataset` con un tipo genérico `<T>`. Esto implica cambiar la declaración de clase a `public class Dataset<T>` y reemplazar todos los usos específicos de `Game` por el tipo `T`. De esta forma, al instanciar la clase se puede especificar el tipo concreto, como por ejemplo `Dataset<Game>` o `Dataset<String>`, según sea necesario.

- **¿Qué funcionalidades del lenguaje Java permiten esto?**

  Java incorpora el uso de *Generics*, una característica del lenguaje que permite escribir clases, interfaces y métodos que operan sobre tipos parametrizados. Esto permite crear estructuras de datos más flexibles y seguras en tiempo de compilación, evitando conversiones explícitas y errores de tipo.

- **¿Qué beneficios trae el uso de `generics` en este tipo de estructuras?**

  El uso de generics mejora la reutilización del código, al permitir que una misma clase funcione con distintos tipos sin duplicación. Además, incrementa la seguridad de tipo al detectar errores en tiempo de compilación. También mejora la legibilidad y mantenimiento del código, facilitando su uso en distintos contextos sin alterar la lógica interna.

9. Conclusión

Desarrollar este laboratorio permitió aplicar los conocimientos adquiridos sobre estructuras de datos y algoritmos de ordenamiento y búsqueda vistos en cátedra. A través de clases como `Game` y `Dataset`, se trabajó con listas dinámicas (`ArrayList`) y se implementaron distintos algoritmos, lo que facilitó la comprensión de su funcionamiento y rendimiento en función del tamaño y tipo de datos.

Entre las ventajas de usar listas enlazadas está su eficiencia al manejar grandes volúmenes de información y es más fácil para aplicar operaciones como filtrado, ordenamiento y búsqueda. Algoritmos como `mergeSort`, `quickSort` y `countingSort` mostraron un mejor rendimiento frente a métodos más simples como `bubbleSort` o `selectionSort`, especialmente en *datasets* grandes.

También se identificaron limitaciones, como la menor eficiencia de algoritmos cuadráticos y que se deben de ordenar antes los datos para realizar búsqueda binaria. Gracias a la realización del laboratorio es más fácil decidir qué estructuras y métodos usar dependiendo de la situación.

En resumen, se cumplieron los objetivos planteados, ya que se logró aplicar y comparar distintos algoritmos en un sistema, utilizando datos del mundo de los videojuegos. Esta experiencia hizo que comprendiéramos temas fundamentales y entregó herramientas útiles para resolver problemas relacionados con la organización y búsqueda de información.