



UNIVERSIDAD DIEGO PORTALES

**Laboratorio 3:  
Algoritmos de ordenamiento y búsqueda en  
Listas enlazadas**

ESTRUCTURAS DE DATOS Y ALGORITMOS  
2025-1

Profesores:  
Valentina Aravena  
Cristián Llull  
Marcos Fantoval

## 1. Introducción

En el mundo digital actual, la gestión y clasificación de información es esencial, especialmente en sectores como el de los videojuegos, donde existen miles de títulos con diversos atributos que deben ser organizados, comparados y analizados constantemente. Ya sea para plataformas de distribución, tiendas digitales o motores de recomendación, la **capacidad de ordenar juegos por precio, categoría o nivel de calidad**, así como **buscar de manera eficiente**, se vuelve fundamental.

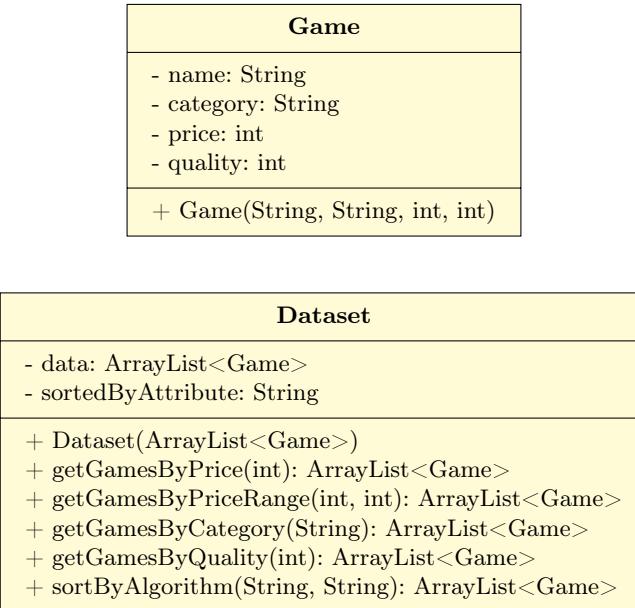
Desde los inicios de la computación, se ha dedicado un porcentaje considerable de los recursos de procesamiento a tareas de ordenamiento de datos. Aunque el hardware ha mejorado notablemente, los algoritmos eficientes siguen siendo clave para optimizar el rendimiento. En este laboratorio, los estudiantes experimentarán de primera mano las **ventajas, desventajas y diferencias prácticas entre diversos algoritmos de ordenamiento y búsqueda**, incluyendo sus tiempos de ejecución y comportamiento en datasets reales simulados.

La temática gira en torno a una clase `Game`, que representa un videojuego mediante atributos relevantes como su **nombre, categoría, precio, y calidad**. A través de esta estructura, se pondrán en práctica conceptos fundamentales de estructuras de datos, tales como el uso de listas enlazadas (`ArrayLists`), algoritmos de ordenamiento (como `mergeSort` o `quickSort`), y búsqueda (lineal y binaria).

## 2. Implementación

En esta sección, desarrollarás una serie de métodos distribuidos en dos clases principales: `Game` y `Dataset`. Estas clases permitirán representar un conjunto de videojuegos y realizar sobre ellos operaciones de búsqueda y ordenamiento, fundamentales en el estudio de estructuras de datos y algoritmos.

### 2.1. Diagramas UML



### 2.2. Especificación de las clases

A continuación se encuentran las clases y sus respectivos métodos y atributos especificados.

#### 2.2.1. Clase Game

La clase `Game` representa un videojuego con **los siguientes atributos**:

- **`name (String)`**: Nombre del juego.
- **`category (String)`**: Categoría o género del juego (por ejemplo: “Aventura”, “Estrategia”, “Acción”).
- **`price (int)`**: Precio del juego en pesos chilenos.
- **`quality (int)`**: Valor de calidad del juego (entre 0 y 100). Este valor puede representar un promedio de calificaciones otorgadas por usuarios o reviewers especializados.

**Métodos a implementar:**

- **`Game(String name, String category, int price, int quality)`**: Constructor de la clase `Game`. Inicializa los atributos del objeto con los valores entregados como parámetros.

#### 2.2.2. Clase Dataset

La clase `Dataset` contiene una estructura de datos dinámica (`ArrayList<Game>`) que almacena una colección de objetos `Game`. Esta clase será la base para experimentar con algoritmos de ordenamiento, búsqueda y para analizar su rendimiento empírico.

**Atributo principal:**

- **data** (`ArrayList<Game>`): Lista de videojuegos.
- **sortedByAttribute** (`String`): Indica por cuál campo está ordenado actualmente el dataset (`precio`, `categoria` o `quality`).

**Métodos a implementar:**

- **Dataset(ArrayList<Game>data)**: Constructor de la clase Dataset. Inicializa el atributo data con el valor recibido como parámetro.
- **getGamesByPrice(int price)**: retorna una `ArrayList<Game>` con todos los juegos cuyo precio sea igual al parámetro entregado.
  - Si el dataset está ordenado por `precio`, se debe utilizar **búsqueda binaria**.
  - En caso contrario, se debe usar **búsqueda lineal**.
- **getGamesByPriceRange(int lowerPrice, int higherPrice)**: retorna todos los juegos cuyo precio esté dentro del rango `[lowerPrice, higherPrice]`.
  - Si el dataset está ordenado por `precio`, se puede aplicar una variación de **búsqueda binaria**.
  - Si no lo está, se debe utilizar **búsqueda lineal**.
- **getGamesByCategory(String category)**: retorna todos los juegos cuya categoría coincida exactamente con el valor del parámetro.
  - Se debe usar **búsqueda binaria** si los datos están ordenados por `categoría`.
  - Si no están ordenados, aplicar **búsqueda lineal**.
- **getGamesByQuality(int quality)**: retorna todos los juegos con nivel de calidad exactamente igual al parámetro recibido.
  - Si el dataset está ordenado por `calidad`, usar **búsqueda binaria**.
  - Si no, usar búsqueda lineal.
- **sortByAlgorithm(String algorithm, String attribute)**: este método permite ordenar el dataset con base en un algoritmo de ordenamiento y un atributo específico.
  - El parámetro `algorithm` puede tomar los siguientes valores: “`bubbleSort`”, “`insertionSort`”, “`selectionSort`”, “`mergeSort`”, “`quickSort`”.
  - Si el valor del parámetro no coincide con ninguno de los anteriores, se debe usar `Collections.sort()`.
  - El parámetro `attribute` determina el campo por el cual ordenar: puede ser “`price`”, “`category`” o “`quality`”.
  - Si no se reconoce el atributo, el ordenamiento se debe hacer por defecto a base de `price`.
  - No olvide actualizar `sortedByAttribute`.

### 3. Experimentación

La experimentación de este laboratorio se basa en haber completado correctamente la implementación de las clases `Game` y `Dataset`. El objetivo de esta sección es **generar datos de prueba aleatorios**, aplicar sobre ellos los **algoritmos de búsqueda y ordenamiento**, y luego **medir y analizar su rendimiento empírico**.

#### 3.1. Generación de Datos

Deberás crear una clase llamada `GenerateData` que permita generar de manera aleatoria una lista de videojuegos (`ArrayList<Game>`) de tamaño N. Los atributos de cada juego deben ser asignados aleatoriamente, respetando las siguientes reglas:

##### 3.1.1. Reglas para `name`

No hay restricciones estrictas para el nombre del juego. Puede generarse combinando al azar palabras desde un arreglo de términos relacionados con juegos, nombres de fantasía o títulos inventados. Por ejemplo:

```
String[] palabras = {"Dragon", "Empire", "Quest", "Galaxy", "Legends", "Warrior"}
```

Un nombre se puede formar tomando dos palabras al azar y concatenándolas, como “GalaxyQuest”.

##### 3.1.2. Reglas para `category`

Debe elegirse una categoría real del mundo de los videojuegos. Puedes seleccionar al azar desde un arreglo como:

```
String[] categorias = {"Acción", "Aventura", "Estrategia", "RPG", "Deportes", "Simulación"}
```

##### 3.1.3. Reglas para `price`

Debe ser un número entero aleatorio entre 0 y 70.000 pesos. Esta gama cubre desde juegos económicos hasta títulos AAA.

##### 3.1.4. Reglas para `quality`

Debe ser un número entero aleatorio entre 0 y 100. Este valor representa una puntuación agregada de calidad percibida, típicamente basada en reseñas o evaluaciones.

##### 3.1.5. Tareas específicas

1. Implementar la clase `GenerateData` con un método que retorne una `ArrayList<Game>` de tamaño N con datos aleatorios válidos.
2. Generar y guardar en archivos separados tres conjuntos de datos con los siguientes tamaños:
  - Tamaño  $10^2$  (100 elementos)
  - Tamaño  $10^4$  (10,000 elementos)
  - Tamaño  $10^6$  (1,000,000 elementos)
3. Estos archivos serán utilizados posteriormente para realizar pruebas de rendimiento. Se recomienda guardar cada conjunto en formato .csv o .txt según tu preferencia.

**Recordar:** Todas las clases deben contar con un método `main` que pruebe el correcto funcionamiento de los métodos creados anteriormente. Pregunte si tiene dudas.

## 3.2. Benchmarks

### 3.2.1. Medición del tiempo de ordenamiento

Usando la clase `Dataset` y los conjuntos de datos generados, debes medir el tiempo de ejecución necesario para ordenar por cada uno de los atributos: `category`, `price`, y `quality`. Los métodos a utilizar son `bubbleSort`, `insertionSort`, `selectionSort`, `mergeSort`, `quickSort` y `collectionsSort`. Cada método se debe ejecutar al menos **3 veces y reportar el promedio** correspondiente. Los resultados de los promedios deben registrarse en una tabla como la siguiente (una tabla por atributo):

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

Algoritmo	Tamaño del dataset	Tiempo (milisegundos)
bubbleSort	$10^2$	
bubbleSort	$10^4$	
bubbleSort	$10^6$	
insertionSort	$10^2$	
...	...	
collections sort	$10^6$	

**Importante:**

- Si el tiempo de ejecución de un algoritmo supera los 300 segundos, puedes interrumpirlo y anotar «más de 300 segundos».
- Realiza una tabla independiente para cada atributo (`category`, `price`, y `quality`).

### 3.2.2. Medición del tiempo de búsqueda

Utilizando solamente el dataset de tamaño  $10^6$ , mide el tiempo de ejecución de cada método `getX` tanto con búsqueda lineal como con búsqueda binaria (según corresponda). Completa los resultados en una tabla como esta:

Cuadro 2: Tiempos de ejecución de búsqueda

Método	Algoritmo	Tiempo (milisegundos)
getGamesByPrice	linearSearch	
getGamesByPrice	binarySearch	
getGamesByPriceRange	linearSearch	
getGamesByPriceRange	binarySearch	
getGamesByCategory	linearSearch	
getGamesByCategory	binarySearch	

No olvide que para la búsqueda binaria es necesario que los datos estén ordenados. A su vez, usted implementó el manejo de `sortedByAttribute` para saber por qué atributo está ordenado el dataset.

### 3.2.3. Creación de gráficos

Deberá crear un gráfico comparativo para cada una de las tablas solicitadas.

## 4. Análisis

Una vez completadas las implementaciones, generación de datos y mediciones de rendimiento, deberás responder y desarrollar las siguientes preguntas y tareas de análisis. Estas actividades te permitirán reflexionar críticamente sobre el comportamiento empírico de los algoritmos implementados y reforzar conceptos clave de eficiencia, aplicabilidad y diseño de estructuras de datos reutilizables.

### 4.1. Comparación entre Búsqueda Lineal y Binaria

En base a la experimentación realizada con los métodos `getGamesByPrice`, `getGamesByCategory`, y `getGamesByQuality`, escribe un breve párrafo comparativo entre búsqueda lineal y búsqueda binaria.

Tu análisis debe incluir:

- Las diferencias principales en su funcionamiento y complejidad temporal.
- Un contraste claro entre los resultados empíricos obtenidos.
- Una reflexión sobre la siguiente pregunta:  
¿En qué escenario crees que es mejor utilizar búsqueda lineal en vez de búsqueda binaria, incluso si esta última tiene mejor complejidad teórica?

### 4.2. Implementación de Counting Sort

Investiga cómo funciona el algoritmo **Counting Sort** y realiza una implementación que permita ordenar el dataset por el atributo *quality*. Este atributo es ideal para esta técnica, ya que es un entero con un rango acotado entre 0 y 100. Luego, completa lo siguiente:

- Mide el tiempo de ejecución de Counting Sort sobre los datasets de tamaño  $10^2$ ,  $10^4$  y  $10^6$ .
- Compara sus tiempos contra los algoritmos anteriores utilizados para ordenar por `quality`.
- Redacta un párrafo con tus observaciones sobre el **rendimiento de Counting Sort**, incluyendo ventajas, limitaciones y contexto de aplicabilidad.

Nota: Documenta brevemente tu implementación y cómo la integraste a la clase `Dataset`.

### 4.3. 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`).

En un párrafo claro y conciso, responde:

- ¿Cómo se podría modificar la clase `Dataset` para que funcione con cualquier tipo de objeto, y no sólo con `Game`?
- ¿Qué funcionalidades del lenguaje Java permiten esto?
- ¿Qué beneficios trae el uso de generics en este tipo de estructuras?