



udp UNIVERSIDAD
DIEGO PORTALES

UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS &
ANÁLISIS DE ALGORITMOS

Laboratorio 3: Algoritmos de ordenamiento y búsqueda de listas enlazadas

Autores:

Cinthya Fuentealba Bravo

Joaquín Roco Fuenzalida

Profesor:

Marcos Fantoal

28 de Mayo 2025

Índice

1. Introducción	2
2. Implementación	3
2.1. Diagramas UML	3
2.2. Especificación de las clases	3
2.2.1. Clase Game	4
2.2.2. Clase Dataset	4
3. Experimentación	6
3.1. Generación de datos	6
3.2. Benchmarks	6
3.2.1. Medición del tiempo de ordenamiento	6
4. Análisis	13
4.1. Comparación entre Búsqueda Lineal y Binaria	13
4.2. Implementación de Counting Sort	13
4.2.1. Ventajas	14
4.2.2. <i>Limitaciones</i>	14
4.3. Uso de Generics en Java para estructuras reutilizables	14
5. Conclusión	15

1. Introducción

En la actualidad, en muchas y diversas áreas industriales, se requiere que la gestión de información o de datos sea eficiente. Una de ellas es el mundo de los videojuegos. En este informe se abordará este último ámbito.

La problemática presentada en esta instancia consiste en generar, mediante codificación Java, una solución para poder tener de una manera más sencilla la capacidad de ordenar juegos según su precio, categoría o el nivel de calidad, dependiendo de lo que el usuario requiera. Además de la organización de los videojuegos, también es importante poder buscarlos de manera eficiente.

Debido a la gran cantidad de títulos disponibles, es indispensable tener mecanismos o herramientas que permitan organizar, buscar y analizar la información de forma rápida y precisa. Por ello, este laboratorio tiene como objetivo implementar y comparar diferentes algoritmos de ordenamiento y de búsqueda sobre una colección de objetos, los cuales representan videojuegos. Por consiguiente, en este trabajo se definen dos clases principales, las cuales son: **Game** y **Dataset**. Estas clases hacen referencia, respectivamente, a los atributos de los videojuegos y a una lista de juegos. Por otra parte, se agrega una tercera clase llamada **GenerateData**, que está encargada de generar datos de manera aleatoria y así simular escenarios reales.

En el transcurso de este laboratorio se valorará el comportamiento de los algoritmos implementados. Para ello se realizará una medición de tiempos de ejecución y análisis de las ventajas y desventajas. Finalmente, se analizará la reutilización de datos por medio del uso de *generics* en Java. Este término hace referencia a una característica que permite definir clases, interfaces y métodos, que trabajen con diversos tipos de datos sin necesidad de escribir versiones separadas para cada uno.

Por ende, dentro del informe se detallan la realización del código que busca dar solución al problema propuesto, también sobre los componentes de este y su funcionamiento. En el siguiente repositorio de GitHub se encuentra la información completa de este laboratorio: https://github.com/Jrgitx/Informe_Lab3_S2. En el enlace se encuentra, el código **.Java** y el archivo **L^AT_EX** para su análisis.

2. Implementación

En esta sección del informe se presentan las principales clases implementadas en este laboratorio, las clases implementadas son **Game** y **Dataset**. Estas clases son desarrolladas en Java. Estas utilizan listas dinámicas denominadas como **ArrayList**, este mecanismo ayuda a realizar operaciones de ordenamiento y búsqueda en un grupo de videojuegos, las cuales son esenciales en el estudio de estructuras de datos y algoritmos.

2.1. Diagramas UML

En esta parte del laboratorio se presentan los diagramas UML, que hacen referencias a las clases creadas en el trabajo.

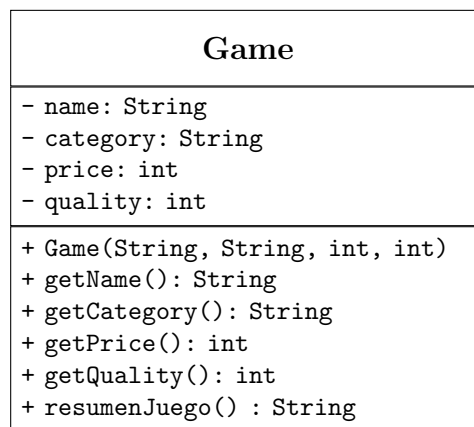


Figura 1: Diagrama UML de la clase Game

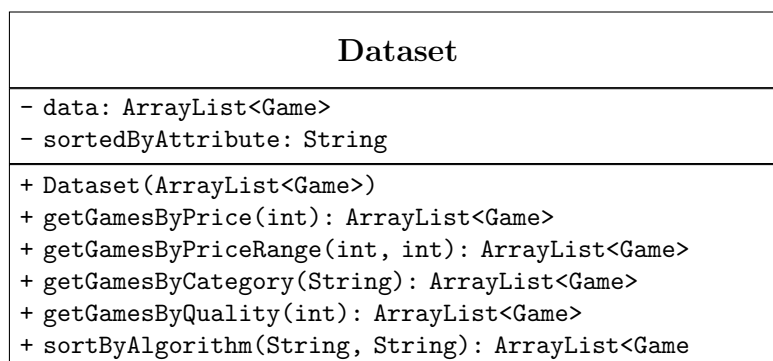


Figura 2: Diagrama UML de la clase Dataset

2.2. Especificación de las clases

En esta parte del informe se encuentran ambas clases, con sus respectivos métodos y atributos de manera específica.

2.2.1. Clase Game

Esta clase representa un videojuego, que posee atributos y métodos, los que se detallan a continuación:

Atributos

- `name(String)`: Este corresponde al nombre del juego.
- `category(String)`: Representa la categoría o género del juego.
- `price(int)`: Es el precio del juego en pesos chilenos.
- `quality(int)`: Este corresponde al valor de la calidad del juego, este valor puede estar entre el 0 y el 100. Este valor puede representar un promedio de calificaciones otorgadas por usuarios.

Métodos

Se implementa un constructor, el cual inicializa los atributos y también se implementan los métodos: `getters`, el cual devuelve el valor del atributo correspondiente, y, por otro lado, `resumenJuego()`, para imprimir la información del juego de manera legible.

2.2.2. Clase Dataset

Esta clase posee una estructura de datos dinámica (`ArrayList<Game>`), la cual almacena una colección de objetos `Game`. Esta clase permite realizar sobre ellos diferentes operaciones de ordenamiento y búsqueda.

Atributos

- `data(ArrayList<Game>)`: Corresponde a una lista dinámica que almacena videojuegos.
- `sortedByAttribute(String)`: Indica el atributo por el cual se encuentra ordenada la lista, ya sea por `price`, `category` o `quality`.

Métodos

- `Dataset(ArrayList<Game>data)`: Constructor que guarda la lista de juegos que es entregada como parámetro en el atributo `data`. Inicializa el atributo `data` con el valor recibido como parámetro.
- `getGamesByPrice(int price)`: Este devuelve una (`ArrayList<Game<data>`) de los juegos cuyo precio es igual al valor indicado.

-
- `getGamesByPriceRange(int lowerPrice, int HigherPrice)`: Retorna todos los juegos cuyo valor en pesos chilenos se encuentre dentro de los parámetros entregados.
 - `getGamesByCategory(String)`: Entrega los juegos que pertenecen a la categoría señalada.
 - `getGamesByQuality(int quality)`: Se filtra y retorna los juegos que tienen la calidad especificada.
 - `sortByAlgorithm(String algorithm, String attribute)`: Este método ordena la lista dataset con base a un algoritmo de ordenamiento, este puede ser (bubblesort, quickSort, entre otros; y también según el atributo elegido.

Depende del atributo por el cual se esté ordenando el grupo de videojuegos, se utilizara una búsqueda binaria o lineal, así se mejorará la eficiencia.

3. Experimentación

Una vez que se encuentran implementadas las clases **Game** y **Dataset**. En esta sección del informe, se generan datos de prueba aleatorios. En ellos se aplican algoritmos de búsqueda y ordenamiento, para posteriormente medir y analizar su rendimiento.

3.1. Generación de datos

Se crea una clase llamada **GenerateData**, la cual es implementada con el objetivo de generar una lista de videojuegos con datos aleatorios. Esta clase logra crear **datasets** de diversos tamaños. Esta clase se utiliza para evaluar el rendimiento de los algoritmos de búsqueda y ordenamiento implementadas en la clase **datasets**.

Para lograr el objetivo de este laboratorio, se genera **datasets** de prueba con tamaño de 10^2 (100 elementos), 10^4 (10.000 elementos) y 10^6 (1.000.000 elementos).

Esta clase sigue las siguientes reglas:

- **Reglas para name:** No existen restricciones estrictas para el nombre del juego. Este se puede crear mediante combinaciones al azar desde palabras que tengan relación con juegos, nombres de fantasías, entre otros. Un nombre puede estar conformado por dos palabras al azar y enlazarlas entre sí.
- **Reglas para category:** La categoría debe ser elegida de una real, es decir, no una palabra al azar en relación con el juego, sino que pertenecer a una categoría del mundo real de los videojuegos.
- **Reglas para price:** Los valores para este número están restringidas entre los valores de 0 y 70.000 pesos chilenos.
- **Reglas para quality:** Corresponde a un valor que puede ir desde el 0 hasta el 100. Representa una puntuación agregada de calidad percibida por los usuarios.

Además de las reglas mencionadas, también se debe implementar la clase con un método que retorne una **ArrayList<Game>**, de tamaño **N**, con datos aleatorios y que sean válidos.

3.2. Benchmarks

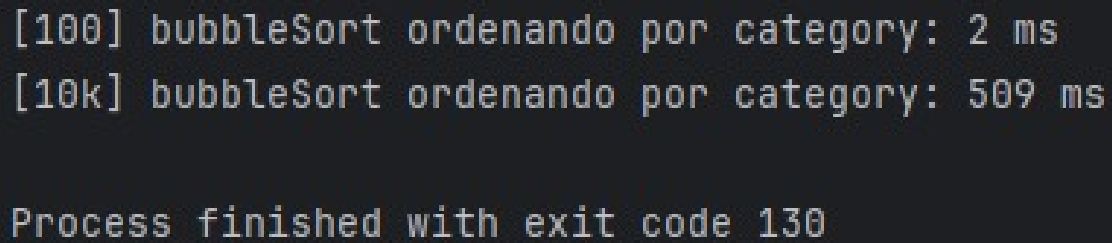
3.2.1. Medición del tiempo de ordenamiento

Con los **datasets** generados en la parte anterior del informe, se realizan pruebas para poder evaluar el tiempo de ejecución en diferentes algoritmos de ordenamiento, estos fueron aplicados a los atributos **category**, **price** y **quality**. Los algoritmos implementados en este laboratorio son:

- **bubbleSort.**
- **insertionSort.**

-
- `selectionSort`.
 - `mergeSort`.
 - `quickSort`.
 - `collectionsSorts`.

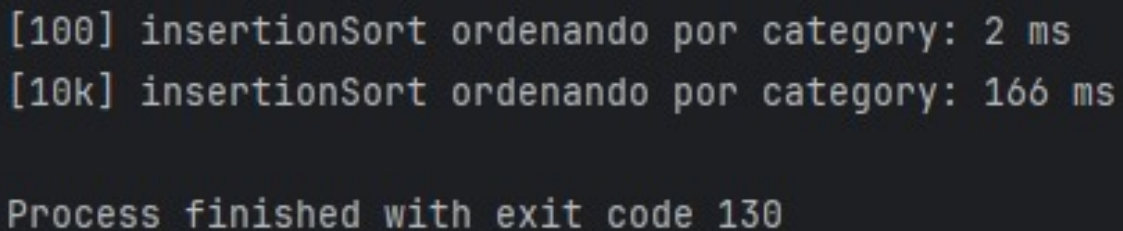
Cada uno de los métodos mencionados se ejecutan tres veces y se reporta el promedio correspondiente. Estos se encuentran registrados en los próximos 3 diagramas a continuación, que se encuentran separados por atributo en cada uno de ellos:



```
[100] bubbleSort ordenando por category: 2 ms
[10k] bubbleSort ordenando por category: 509 ms

Process finished with exit code 130
```

Figura 3: Captura de tiempo de ordenamiento con el método `bubbleSort` en atributo `category`



```
[100] insertionSort ordenando por category: 2 ms
[10k] insertionSort ordenando por category: 166 ms

Process finished with exit code 130
```

Figura 4: Captura de tiempo de ordenamiento con el método `insertionSort` en atributo `category`

Algoritmo	Tamaño del dataset	Tiempo(milisegundos)
bubbleSort	10^2	2
bubbleSort	10^4	300 <
bubbleSort	10^6	
insertionSort	10^2	2
insertionSort	10^4	166
insertionSort	10^6	
selectionSort	10^2	
selectionSort	10^4	
selectionSort	10^6	
mergeSort	10^2	
mergeSort	10^4	
mergeSort	10^6	
quickSort	10^2	
quickSort	10^4	
quickSort	10^6	
collectionsSorts	10^2	
collectionsSorts	10^4	
collectionsSorts	10^6	

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

```
[100] bubbleSort ordenando por price: 2 ms
[10k] bubbleSort ordenando por price: 342 ms

Process finished with exit code 130
```

Figura 5: Captura de tiempo de ordenamiento con el método `bubbleSort` en atributo `price`

```
[100] insertionSort ordenando por price: 1 ms  
[10k] insertionSort ordenando por price: 129 ms  
  
Process finished with exit code 130
```

Figura 6: Captura de tiempo de ordenamiento con el método `insertSort` en atributo `price`

```
[100] selectionSort ordenando por price: 1 ms  
[10k] selectionSort ordenando por price: 186 ms  
  
Process finished with exit code 130
```

Figura 7: Captura de tiempo de ordenamiento con el método `selectionSort` en atributo `price`

Algoritmo	Tamaño del dataset	Tiempo(milisegundos)
bubbleSort	10^2	2
bubbleSort	10^4	300<
bubbleSort	10^6	
insertionSort	10^2	1
insertionSort	10^4	129
insertionSort	10^6	
selectionSort	10^2	1
selectionSort	10^4	166
selectionSort	10^6	
mergeSort	10^2	
mergeSort	10^4	
mergeSort	10^6	
quickSort	10^2	
quickSort	10^4	
quickSort	10^6	
collectionsSorts	10^2	
collectionsSorts	10^4	
collectionsSorts	10^6	

Cuadro 2: Tiempos de ejecución de ordenamiento para el atributo price

```
[100] bubbleSort ordenando por quality: 2 ms
[10k] bubbleSort ordenando por quality: 337 ms

Process finished with exit code 130
```

Figura 8: Captura de tiempo de ordenamiento con el método bubbleSort en atributo quality

```
[100] insertionSort ordenando por quality: 1 ms  
[10k] insertionSort ordenando por quality: 134 ms  
  
Process finished with exit code 130
```

Figura 9: Captura de tiempo de ordenamiento con el método `insertSort` en atributo `quality`

```
[100] selectionSort ordenando por quality: 1 ms  
[10k] selectionSort ordenando por quality: 186 ms  
  
Process finished with exit code 130
```

Figura 10: Captura de tiempo de ordenamiento con el método `selectionSort` en atributo `quality`

Algoritmo	Tamaño del dataset	Tiempo(milisegundos)
bubbleSort	10^2	2
bubbleSort	10^4	300 <
bubbleSort	10^6	
insertionSort	10^2	1
insertionSort	10^4	134
insertionSort	10^6	
selectionSort	10^2	1
selectionSort	10^4	186
selectionSort	10^6	
mergeSort	10^2	
mergeSort	10^4	
mergeSort	10^6	
quickSort	10^2	
quickSort	10^4	
quickSort	10^6	
collectionsSorts	10^2	
collectionsSorts	10^4	
collectionsSorts	10^6	

Cuadro 3: Tiempos de ejecución de ordenamiento para el atributo `quality`

4. Análisis

4.1. Comparación entre Búsqueda Lineal y Binaria

En la sección anterior (experimentación), se realizan pruebas en los métodos `getGamesByPrice`, `getGamesByCategory` y `getGamesByQuality`. En ella se aprecia que existe una gran diferencia en los tiempos de ejecución. Estos tiempos son entre la búsqueda lineal y la búsqueda binaria.

Los tiempos de ejecución de los dos tipos de búsqueda mencionados se diferencian principalmente en que, por una parte, la búsqueda lineal recorre la lista elemento por elemento, hasta encontrar el valor que se está buscando, con una complejidad temporal de $O(n)$. Por el contrario, la búsqueda binaria requiere que los datos estén ordenados con anterioridad, luego realiza una división en mitades de manera sucesiva, con esto se logra un tiempo de $O(\log n)$.

A partir de los resultados obtenidos se logra observar que la experimentación muestra que la búsqueda binaria es significativamente más rápida que la búsqueda lineal, sobre todo implementada en listados con grandes cantidades de datos, sin embargo, los datos de la búsqueda binaria necesitan que se encuentren ordenados previamente. Es una búsqueda eficiente ya que reduce el espacio de búsqueda a la mitad en cada paso.

No obstante, la búsqueda lineal sigue siendo utilizada en ciertos contextos, como por ejemplo: cuando la cantidad de datos es pequeña, cuando no se conoce el orden, cuando ordenar antes de comenzar a buscar es más alto, entre otros. Por lo tanto, la búsqueda lineal puede ser más conveniente en contextos donde el escenario es simple o sin orden, ya que tiene menor requerimiento y preparación previa, a pesar de que la búsqueda binaria es más eficiente teóricamente.

Por último, se concluye que la conveniencia de utilizar búsqueda lineal o binaria, va a depender directamente del contexto en el que se encuentre, también del tipo y cantidad de información con la que se desea trabajar. Cada uno de los tipos de búsqueda tienen ventajas y desventajas, y hay que saber adecuar cuál de estos dos métodos utilizar según la necesidad.

4.2. Implementación de Counting Sort

Counting Sort es un algoritmo de ordenamiento no comparativo, ya que realiza una cuenta de las ocurrencias y las organiza según su frecuencia. Es implementado solamente para el atributo `quality`, ya que es un atributo que tiene un rango fijo y reducido (desde 0 a 100). Los atributos `price` y `category`, tienen características que los hacen inapropiados para este tipo de ordenamiento, porque tienen un amplio rango y datos textuales.

Como resultado, se obtiene que el ordenamiento mediante Counting Sort es más

rápido que los algoritmos como `bubbleSort` y `quickSort`, esto cuando se emplea sobre el atributo `quality`. Este tipo de ordenamiento mejora el rendimiento ya que su complejidad es lineal $O(n+k)$, donde n corresponde al número de elementos que se desean ordenar y k pertenece al rango de valores posibles del atributo.

4.2.1. Ventajas

- Tiempo de ejecución $O(n+k)$, es decir, con un tiempo óptimo si k no es un número grande. Complejidad lineal.
- No realiza comparaciones, por lo que es eficiente en atributos con rango limitado.
- Es estable, ya que mantiene el orden relativo en elementos de igual valor.

4.2.2. Limitaciones

- No recomendable aplicar sobre atributos que poseen un rango amplio.
- No funciona correctamente con atributos que sean del tipo `String`.
- Requiere espacio adicional proporcional a k .

Este tipo de ordenamiento fue implementado en la clase `Dataset`, dentro del método `sortByAlgorithm(String algorithm, String attribute)`. Esto permite que cuando el usuario especifique “countingSort”, como nombre del algoritmo, se ejecuta este tipo de ordenamiento en vez de los otros tipos de ordenamiento.

4.3. Uso de Generics en Java para estructuras reutilizables

En el desarrollo del laboratorio, la clase `Dataset` fue diseñada específicamente para tener objetos de tipo `Game`. Sin embargo, es mejor la utilización de otras estructuras de datos que funcionen con distintos tipos de objetos. Es por ello, que Java entrega una característica conocida como *generics*.

El uso de *generics*, permite que una clase pueda trabajar con cualquier tipo de dato, esto sin perder seguridad de tipos en tiempos de compilación. Esto puede implementarse en la clase `Dataset`, para que funcione con cualquier tipo de objeto. Se debe reemplazar el tipo específico (en este caso el tipo `Game`), por un parámetro genérico, como, por ejemplo: `<T>` en la parte de definición de la clase. De esta forma, `Dataset<T>`, podría almacenar objetos de cualquier tipo que el usuario le indique.

El principal beneficio de aplicar *generics* en estructuras como `Dataset` es que se logra un código más reutilizable, flexible y seguro, se puede utilizar la misma clase para diferentes tipos de datos sin necesidad de duplicar la lógica, y se evitan errores de conversión de tipos al momento de ejecutar el programa.

5. Conclusión

Durante el desarrollo de este laboratorio se exploraron diferentes técnicas de ordenamiento y búsqueda aplicadas sobre una estructura de datos dinámica, la cual almacena objetos en representación de los videojuegos. También se realizaron las clases necesarias para organizar, acceder a la información de manera eficiente, y evaluar el comportamiento de los diferentes algoritmos.

Como resultado obtenido de todo el trabajo se permite observar diferencias en el rendimiento entre los métodos de ordenamiento. Se comprobó que el uso del algoritmo *Counting Sort* es eficiente en algunos casos y en otros en los que no es recomendable su uso, sobre todo en atributos de un alto rango o en tipos de datos de tipo *String*.

Por otra parte, se aprendió sobre los métodos de búsqueda, y se evidenció que la búsqueda binaria supera a la lineal si se prioriza eficiencia, siempre y cuando los datos se encuentren ordenados. Es importante decidir qué método de búsqueda es el más adecuado según el contexto en el que nos encontremos.

Finalmente, se destacó la importancia del uso de estructuras genéricas en Java para mejorar la reutilización del código y la adaptabilidad de las clases. Esta experiencia permitió aplicar de forma práctica los conceptos teóricos del curso y reflexionar sobre las decisiones de diseño que impactan directamente en el rendimiento de las aplicaciones.