UNIVERSIDAD DIEGO PORTALES
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES

ESTRUCTURAS DE DATOS &
ANÁLISIS DE ALGORITMOS

Laboratorio 3: Algoritmos de ordenamiento y búsqueda en Listas enlazadas

---

Autores:
Valentina Martínez
José Pablo Peña

Profesor:
Marcos Fantoval

26 de mayo de 2025

Índice

1. Introducción

Hoy en día los videojuegos van en un progreso que los hace poseer variadas características que deben ser organizadas y analizadas, por lo tanto se tienen formas eficientes que permiten ordenar juegos por precio, categoría o calidad, así como buscar títulos específicos de forma rápida y precisa. Este laboratorio es sobre llevar a cabo esto, simulando el trabajo con listas de videojuegos para aplicar estructuras de datos y algoritmos.

El objetivo principal de este laboratorio es analizar algoritmos de ordenamiento y búsqueda, aplicándolos sobre listas enlazadas sobre juegos. Entonces se tiene la clase Game, que posee objetos los cuales son juegos que tienen atributos como nombre, categoría, precio y calidad. Todas las operaciones de búsqueda y ordenamiento se realizarán mediante la clase Dataset, que trabaja de distintas formas dependiendo del atributo seleccionado y del tipo de búsqueda que se necesite.

Para profundizar más, se crea la clase llamada GenerateData, que crea más listas de videojuegos pero con datos aleatorios. Estas listas son utilizadas para aplicar los algoritmos de búsqueda y ordenamiento. Cada videojuego generado tendrá atributos definidos de forma aleatoria, simulando así un entorno similar al de tiendas digitales o plataformas de videojuegos.

Finalmente se medirá el rendimiento de cada algoritmo en función del tiempo que tarda en ejecutarse, dependiendo del tipo de ordenamiento o búsqueda aplicado. Se evaluarán tanto casos donde los datos están ordenados previamente y también casos en los que estén desordenados. Con esto se busca lograr el objetivo de cómo se comportan estos algoritmos en la práctica, identificar sus ventajas y desventajas, y analizar cual conviene usar en distintos escenarios en torno a listas de videojuegos.

Todo el desarrollo de este laboratorio se ha realizado utilizando el lenguaje de programación Java, el cual, al ser un lenguaje orientado a objetos, facilita la implementación de estructuras de datos y el análisis comparativo de los algoritmos. El código completo del proyecto se encuentra disponible en: `https://github.com/valemiauz/Lab-3`

2. Equipos y materiales

Para la realización de este laboratorio y su informe se utilizaron los siguientes materiales y equipos:

- **IntelliJ IDEA:** Es un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos, en este caso fue utilizado para el desarrollo del código.
- **Computador:** Computador común en el cual se ha instalado IntelliJ IDEA previamente para poder trabajar.

- **Github:** Una plataforma web que permite almacenar y gestionar proyectos de programación, en donde se crea un repositorio para subir el laboratorio y su código.
- **Overleaf:** Es una plataforma en línea para escribir, editar y colaborar en documentos científicos y técnicos usando LaTeX, el cual es un sistema de preparación de documentos.

3. Actividades realizadas

A grandes rasgos las actividades realizadas para llevar a cabo el laboratorio han sido desarrollar el sistema para organizar videojuegos usando Java en el entorno IntelliJ IDEA, el cual permitió realizar el programa en pareja, completando lo pedido siendo en general estructuras de datos y algoritmos de ordenamiento, y búsqueda tanto lineal como binaria en listas enlazadas, para luego proceder con la experimentación, después se completó el ánalisis y se escribió este informe solicitado en LaTeX trabajándolo también en colaboración, finalmente se creó un repositorio donde se guardó el código del proyecto junto al pdf del informe.

4. Implementación

A continuación se desarrollan las clases principales: Game y Dataset. Estas clases permitirán representar un conjunto de videojuegos y realizar sobre ellos operaciones de búsqueda y ordenamiento.

4.1. Clase Game

La clase Game tiene como propósito crear objetos de tipo juego con atributos como nombre, categoría, precio y calidad. Para ello, se implementó un constructor con parámetros. También se añadieron métodos getters y setters, ya que los atributos fueron declarados como privados, lo que permite acceder y modificar sus valores de manera controlada.

```java
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 void setName(String name) {
        this.name = name;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public void setQuality(int quality) {
        this.quality = quality;
    }

    public String getName(){
        return name;
    }

    public String getCategory(){
        return category;
    }

    public int getPrice(){
        return price;
    }

    public int getQuality(){
        return quality;
    }
}
```

4.2. Clase Dataset

La clase DataSet tiene como finalidad implementar algoritmos de ordenamiento y búsqueda. Para ello, se declara un arraylist que almacena una colección de objetos Game. Y un atributo de tipo String llamado sortedByAttribute, que indica el criterio por el cual está ordenado el dataset. Además, se implementó un constructor con parámetros.

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

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

4.2.1. Métodos de búsqueda

- **getGamesByPrice(int price):** Este método ordena los juegos según su precio. Si los datos están ordenados por precio, utiliza búsqueda binaria para encontrar cada juego. De lo contrario, se utiliza búsqueda lineal.

```
        public ArrayList<Game> getGamesByPrice(int price) {
        ArrayList<Game> result = new ArrayList<>();

        if ("price".equals(sortedByAttribute)) {
            int left = 0, right = data.size() - 1;

            while (left <= right) {
                int mid = (left + right) / 2;
                int midPrice = data.get(mid).getPrice();

                if (midPrice == price) {
                    int i = mid;
                    while (i >= 0 && data.get(i).getPrice() ==
                        price) {
                        result.add(0, data.get(i));
                        i--;
                    }
                    i = mid + 1;
                    while (i < data.size() &&
                        data.get(i).getPrice() == price) {
                        result.add(data.get(i));
                        i++;
                    }
                    break;
                } else if (midPrice < price) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
```

```
27                  }
28              }
29
30          } else {
31              for (Game g : data) {
32                  if (g.getPrice() == price) {
33                      result.add(g);
34                  }
35              }
36          }
37
38          return result;
39      }
```

- **getGamesByPriceRange(int lowerPrice, int higherPrice):** Busca todos los juegos cuyo precio esté dentro del rango especificado. Se utilizan los métodos auxiliares findFirstIndex() y findLastIndex() para encontrar el rango mediante búsqueda binaria. Si el dataset no está ordenado por precio, se utiliza búsqueda lineal.

```
1         public ArrayList<Game> getGamesByPriceRange(int
              lowerPrice, int higherPrice) {
2         ArrayList<Game> result = new ArrayList<>();
3         if ("price".equals(sortedByAttribute)) {
4             int startIndex = findFirstIndex(lowerPrice);
5             int endIndex = findLastIndex(higherPrice);
6             for (int i = startIndex; i <= endIndex && i <
                  data.size(); i++) {
7                 result.add(data.get(i));
8             }
9
10        } else {
11            for (Game g : data) {
12                int price = g.getPrice();
13                if (price >= lowerPrice && price <=
                      higherPrice) {
14                    result.add(g);
15                }
16            }
17        }
18
19        return result;
20    }
```

- **getGamesByCategory(String category):** Busca todos los juegos según su categoría específica. Si los datos están ordenados por categoría, se utiliza compareTo() para búsqueda binaria y equals() para búsqueda lineal.

```
1   public ArrayList<Game> getGamesByCategory(String category)
        {
2       if (category == null) return new ArrayList<>();
3       ArrayList<Game> result = new ArrayList<>();
4
5       if ("category".equals(sortedByAttribute)) {
6           int left = 0, right = data.size() - 1;
7
8           while (left <= right) {
9               int mid = (left + right) / 2;
10              String midCategory =
                    data.get(mid).getCategory();
11
12              int cmp = midCategory.compareTo(category);
13
14              if (cmp == 0) {
15              int i = mid;
16              while (i >= 0 &&
                    data.get(i).getCategory().equals(category))
                    {
17                      result.add(0, data.get(i));
18                      i--;
19                  }
20              i = mid + 1;
21              while (i < data.size() &&
                    data.get(i).getCategory().equals(category))
                    {
22                      result.add(data.get(i));
23                      i++;
24                  }
25                  break;
26              } else if (cmp < 0) {
27                  left = mid + 1;
28              } else {
29                  right = mid - 1;
30              }
31          }
32      } else {
33          for (Game g : data) {
34              if (g.getCategory().equals(category)) {
35                  result.add(g);
36              }
37          }
38      }
39
40      return result;
41  }
```

- **getGamesByQuality(int quality):** Muestra todos los juegos con un nivel de calidad determinado entre 0 y 100. Utiliza la misma estructura que los métodos anteriores: Si se ordena por calidad se usa búsqueda binaria, de lo contrario búsqueda lineal.

```
        public ArrayList<Game> getGamesByQuality(int quality) {
        ArrayList<Game> result = new ArrayList<>();

        if ("quality".equals(sortedByAttribute)) {
            int left = 0, right = data.size() - 1;

            while (left <= right) {
                int mid = (left + right) / 2;
                int midQuality = data.get(mid).getQuality();

                if (midQuality == quality) {
                    int i = mid;
                    while (i >= 0 && data.get(i).getQuality()
                        == quality) {
                        result.add(0, data.get(i));
                        i--;
                    }
                    i = mid + 1;
                    while (i < data.size() &&
                        data.get(i).getQuality() == quality) {
                        result.add(data.get(i));
                        i++;
                    }
                    break;
                } else if (midQuality < quality) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        } else {
            for (Game g : data) {
                if (g.getQuality() == quality) {
                    result.add(g);
                }
            }
        }

        return result;
    }
```

- **sortByAlgorithm(String algorithm, String attribute):** Ordena el dataset usando el algoritmo especificado para el atributo. Se puede ordenar con Bubble sort (compara elementos adyacentes), Insertion Sort (inserta cada elemento en su posición correcta), Selection Sort (selecciona el mínimo y lo coloca al inicio), Merge Sort (divide y conquista), Quick Sort (cambia a insertion sort para arreglos pequeños) o Counting Sort (creado específicamente para el atributo de calidad). Si no coincide con ningún parámetro, se usa Collection Sort.

```
1         public void sortByAlgorithm(String algorithm, String
              attribute) {
2         String validAttribute;
3         switch (attribute != null ? attribute.toLowerCase() :
              "") {
4             case "price":
5                 validAttribute = "price";
6                 break;
7             case "category":
8                 validAttribute = "category";
9                 break;
10            case "quality":
11                validAttribute = "quality";
12                break;
13            default:
14                validAttribute = "price";
15                break;}
16        switch (algorithm != null ? algorithm.toLowerCase() :
              "") {
17            case "bubblesort":
18                bubbleSort(validAttribute);
19                break;
20            case "insertionsort":
21                insertionSort(validAttribute);
22                break;
23            case "selectionsort":
24                selectionSort(validAttribute);
25                break;
26            case "mergesort":
27                mergeSort(validAttribute);
28                break;
29            case "quicksort":
30                quickSort(validAttribute);
31                break;
32            case "countingsort":
33                countingSort(validAttribute);
34                break;
35            default:
36                collectionsSort(validAttribute);
37                break;
38        }
39        sortedByAttribute = validAttribute;
40    }
```

Bubble Sort

```
        private void bubbleSort(String attribute) {
        int n = data.size();
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (compareGames(data.get(j), data.get(j + 1),
                    attribute) > 0) {
                    Game temp = data.get(j);
                    data.set(j, data.get(j + 1));
                    data.set(j + 1, temp);
                }
            }
        }
    }
```

Inserion Sort

```
    private void insertionSort(String attribute) {
        for (int i = 1; i < data.size(); i++) {
            Game key = data.get(i);
            int j = i - 1;

            while (j >= 0 && compareGames(data.get(j), key,
                attribute) > 0) {
                data.set(j + 1, data.get(j));
                j--;
            }
            data.set(j + 1, key);
        }
    }
```

Selection Sort

```
    private void selectionSort(String attribute) {
        int n = data.size();
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (compareGames(data.get(j),
                    data.get(minIndex), attribute) < 0) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                Game temp = data.get(i);
                data.set(i, data.get(minIndex));
                data.set(minIndex, temp);
            }
        }
    }
```

Merge Sort (con merge y mergeSortHelper)

```
    private void mergeSort(String attribute) {
        if (data.size() <= 1) return;
        mergeSortHelper(0, data.size() - 1, attribute);
    }

    private void mergeSortHelper(int left, int right, String
        attribute) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSortHelper(left, mid, attribute);
            mergeSortHelper(mid + 1, right, attribute);
            merge(left, mid, right, attribute);
        }
    }

    private void merge(int left, int mid, int right, String
        attribute) {
        ArrayList<Game> leftArray = new ArrayList<>();
        ArrayList<Game> rightArray = new ArrayList<>();

        for (int i = left; i <= mid; i++) {
            leftArray.add(data.get(i));
        }
        for (int j = mid + 1; j <= right; j++) {
            rightArray.add(data.get(j));
        }
        int i = 0, j = 0, k = left;

        while (i < leftArray.size() && j < rightArray.size()) {
            if (compareGames(leftArray.get(i),
                rightArray.get(j), attribute) <= 0) {
                data.set(k, leftArray.get(i));
                i++;
            } else {
                data.set(k, rightArray.get(j));
                j++;
            }
            k++;
        }

        while (i < leftArray.size()) {
            data.set(k, leftArray.get(i));
            i++;
            k++;
        }
        while (j < rightArray.size()) {
            data.set(k, rightArray.get(j));
            j++;
            k++;
        }
    }
```

Quick Sort (con quickSortIterative)

```
1     private void quickSort(String attribute) {
2         if (data.size() <= 1) return;
3         if (data.size() <= 10) {
4             insertionSort(attribute);
5             return;
6         }
7         quickSortIterative(0, data.size() - 1, attribute);
8     }
9
10    private void quickSortIterative(int low, int high, String
          attribute) {
11        Stack<Integer> stack = new Stack<>();
12        stack.push(low);
13        stack.push(high);
14
15        while (!stack.isEmpty()) {
16            high = stack.pop();
17            low = stack.pop();
18
19            if (high - low <= 10) {
20                insertionSortRange(low, high, attribute);
21                continue;
22            }
23
24            int pivot = partition(low, high, attribute);
25
26            if (pivot - 1 > low) {
27                stack.push(low);
28                stack.push(pivot - 1);
29            }
30
31            if (pivot + 1 < high) {
32                stack.push(pivot + 1);
33                stack.push(high);
34            }
35        }
36    }
```

Collections Sort

```
1     private void collectionsSort(String attribute) {
2         switch (attribute) {
3         case "price":
4         data.sort(Comparator.comparingInt(Game::getPrice));
5         break;
6         case "category":
7         data.sort(Comparator.comparing(Game::getCategory));
8         break;
9         case "quality":
10        data.sort(Comparator.comparingInt(Game::getQuality));
11        break;
12        }
13    }
```

5. Experimentación

Una vez implementadas correctamente las clases Game y Dataset, el siguiente objetivo consiste en generar datos aleatorios utilizando la clase GenerateData. A continuación, se analiza el funcionamiento de esta clase y después se aplican distintos algoritmos de búsqueda y ordenamiento sobre los datos generados, registrando los tiempos de ejecución para evaluar su rendimiento.

5.1. Generación de datos

La clase GenerateData permite generar de manera aletoria una lista de videojuegos de tamaño N, con atributos aleatorios. Siendo esto necesario para llevar a cabo la experimentación.

```
1     class GenerateData {
2     private ArrayList<Game> data = new ArrayList<>();
3     private Random random = new Random();
4
5     private String[] palabras = {"Dragon", "Empire", "Quest",
          "Galaxy", "Legends", "Warrior"};
6
7     private String[] categorias = {"Accion", "Aventura",
          "Estrategia", "RPG", "Deportes", "Simulacion"};
8
9     public GenerateData() {
10    }
```

El método generateRandomName() genera nombres de una o dos palabras elegidas de forma aleatoria. El método generateRandomCategory() selecciona aleatoriamente una categoría. El método generateRandomPrice() y generateRandomQuality() generan valores numéricos aleatorios para representar el precio (entre 0 y 70.000) y la calidad (entre 0 y 100) de cada juego.

```
1      private String generateRandomName() {
2         if (random.nextBoolean()) {
3             return palabras[random.nextInt(palabras.length)];
4         } else {
5             String palabra1 =
                  palabras[random.nextInt(palabras.length)];
6             String palabra2 =
                  palabras[random.nextInt(palabras.length)];
7             return palabra1 + palabra2;
8         }
9     }
10
11    private String generateRandomCategory() {
12        return categorias[random.nextInt(categorias.length)];
13    }
14
15    private int generateRandomPrice() {
```

```
        return random.nextInt(70001);
    }

    private int generateRandomQuality() {
        return random.nextInt(101);
    }
```

El método createGames(int n) genera una lista de objetos Game con los atributos aleatorios, utilizando los métodos auxiliares aleatorios. Se repite este proceso n veces para crear un conjunto de datos del tamaño solicitado, que luego se almacena en una lista y se retorna.

```
            public ArrayList<Game> createGames(int n) {
        ArrayList<Game> games = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            String name = generateRandomName();
            String category = generateRandomCategory();
            int price = generateRandomPrice();
            int quality = generateRandomQuality();

            Game game = new Game(name, category, price, quality);
            games.add(game);
        }

        return games;
    }
```

El método generateToFile, permite exportar dicha lista a un archivo CSV. Primero crea un archivo con el nombre indicado y escribe una tabla con los datos de los juegos. Después pone los nombres de las columnas (nombre, categoría, precio y calidad), y luego escribe cada juego en una línea.

```

    public void generateToFile(ArrayList<Game> games, String
        filename) {
        try (PrintWriter writer = new PrintWriter(new
            FileWriter(filename))) {
            writer.println("name,category,price,quality");

            for (Game game : games) {
                writer.printf("%s,%s,%d,%d%n",
                        game.getName(),
                        game.getCategory(),
                        game.getPrice(),
                        game.getQuality());
            }

        } catch (IOException e) {
            System.err.println("Error al escribir archivo: " +
                e.getMessage());
```

```
16              }
17          }
18  }
```

5.2. Benchmarks

5.2.1. Medición del tiempo de ordenamiento

A continuación se midió el tiempo promedio necesario para ordenar los datos según los tres atributos: categoría, precio y calidad. Con esta información se completan los cuadros 1, 2 y 3 respectivamente:

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

Algoritmo	**Tamaño del dataset**	**Tiempo (milisegundos)**
bubbleSort	10^2	0,718 ms
bubbleSort	10^4	593,808 ms
bubbleSort	10^6	más de 300 seg
insertionSort	10^2	0,161 ms
insertionSort	10^4	140,172 ms
insertionSort	10^6	más de 300 seg
selectionSort	10^2	0,287 ms
selectionSort	10^4	369,753
selectionSort	10^6	más de 300 seg
mergeSort	10^2	0,139 ms
mergeSort	10^4	7,644 ms
mergeSort	10^6	415,662 ms
quickSort	10^2	0,111 ms
quickSort	10^4	3,202 ms
quickSort	10^6	217,381 ms
collectionsSort	10^2	0,013 ms
collectionsSort	10^4	0,859 ms
collectionsSort	10^6	86,856 ms

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

Algoritmo	**Tamaño del dataset**	**Tiempo (milisegundos)**
bubbleSort	10^2	0,477 ms
bubbleSort	10^4	413,152 ms
bubbleSort	10^6	más de 300 seg
insertionSort	10^2	0,217 ms
insertionSort	10^4	129,980 ms
insertionSort	10^6	más de 300 seg
selectionSort	10^2	0,147 ms
selectionSort	10^4	191,153 ms
selectionSort	10^6	más de 300 seg
mergeSort	10^2	0,201 ms
mergeSort	10^4	6,707 ms
mergeSort	10^6	431,882 ms
quickSort	10^2	0,064 ms
quickSort	10^4	2,897 ms
quickSort	10^6	255,329 ms
collectionsSort	10^2	0,010 ms
collectionsSort	10^4	1,082 ms
collectionsSort	10^6	218,722 ms

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

Algoritmo	**Tamaño del dataset**	**Tiempo (milisegundos)**
bubbleSort	10^2	0,488 ms
bubbleSort	10^4	461,799 ms
bubbleSort	10^6	más de 300 seg
insertionSort	10^2	0,222 ms
insertionSort	10^4	159,671 ms
insertionSort	10^6	más de 300 seg
selectionSort	10^2	0,304 ms
selectionSort	10^4	231,544 ms
selectionSort	10^6	más de 300 seg
mergeSort	10^2	0,154 ms
mergeSort	10^4	6,761 ms
mergeSort	10^6	414,484 ms
quickSort	10^2	0,075 ms
quickSort	10^4	2,405 ms
quickSort	10^6	234,053 ms
collectionsSort	10^2	0,329 ms
collectionsSort	10^4	9,274 ms
collectionsSort	10^6	111,663 ms

5.2.2. Medición del tiempo de búsqueda

Ahora solo utilizando el 10^6, se mide el tiempo de ejecución con búsqueda lineal y binaria en los métodos mostrados en el cuadro 4:

Cuadro 4: Tiempos de ejecución de búsqueda

Método	**Algoritmo**	**Tiempo (milisegundos)**
getGamesByPrice	linearSearch	3,997 ms
getGamesByPrice	binarySearch	0,006 ms
getGamesByPriceRange	linearSearch	13,798 ms
getGamesByPriceRange	binarySearch	7,262 ms
getGamesByCategory	linearSearch	13,975 ms
getGamesByCategory	binarySearch	535,790 ms

5.2.3. Creación de gráficos

Se crean gráficos comparativos para los según los cuatro cuadros realizados anteriormente:

La figura 1, muestra un gráfico de barras que enseña el rendimiento de todos los algoritmos de ordenamiento por atributos.

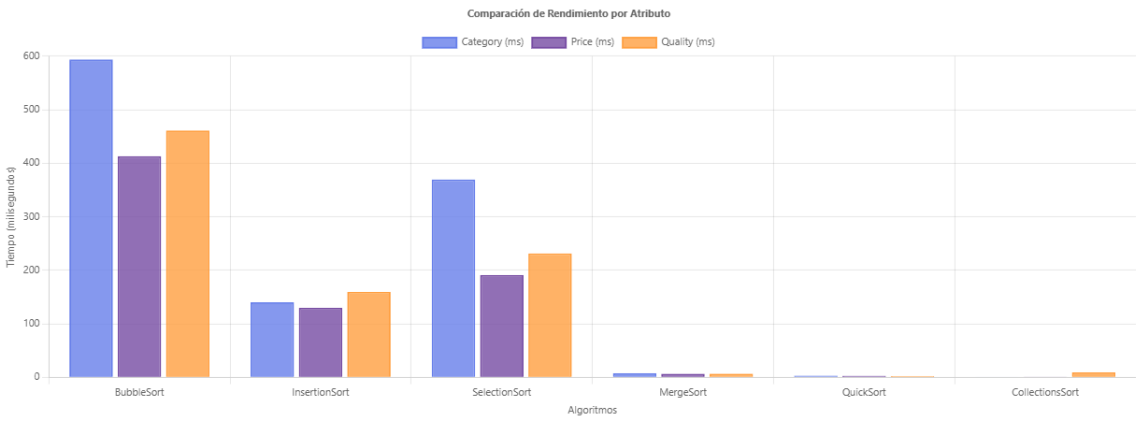

Figura 1: Rendimiento algoritmos por atributos.

La figura 2, muestra un gráfico de barras que muestra la comparación de ejecución rendimiento de los algoritmos de búsqueda siendo estos búsqueda lineal y binaria.

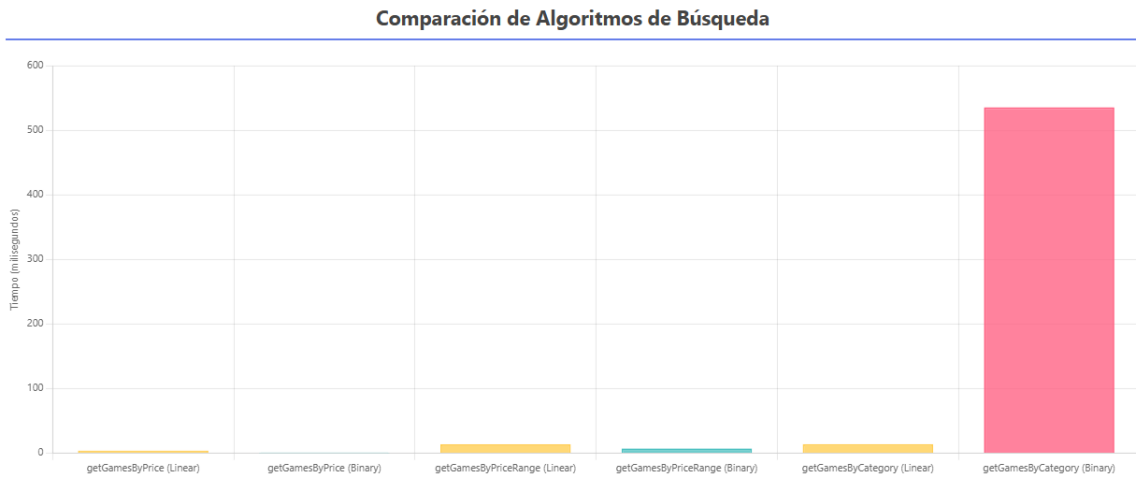

Figura 2: Rendimiento algoritmos.

6. Ánalisis

En esta sección, se interpretarán los resultados obtenidos en la experimentación, proporcionando una comprensión profunda del rendimiento de los algoritmos de ordenamiento y búsqueda en el contexto de la gestión de videojuegos.

6.1. Comparación entre búsqueda lineal y binaria

Para comparar el rendimiento de la búsqueda lineal y la búsqueda binaria, se realizaron experimentos con los métodos getGamesByPrice, getGamesByCategory y getGamesByQuality. En los métodos getGamesByPrice y getGamesByPriceRange, la búsqueda binaria toma menos tiempo que la búsqueda lineal. Esto se debe, según las restricciones del código, a que el dataset está ordenado por precio. Sin embargo, para el método getGamesByCategory la búsqueda lineal es considerablemente más rápida.

¿En qué escenario sería mejor utilizar búsqueda lineal en vez de búsqueda binaria, incluso si esta última tiene mejor complejidad teórica? En este caso la implementación de la búsqueda binaria está haciendo operaciones de mucho rendimiento. Si hay muchos juegos que pertenecen a la misma categoría, después de encontrar uno con búsqueda binaria, el proceso de buscar todos los demás puede volverse casi tan lento como revisar uno por uno. Además, comparar tantas cadenas de texto (‘compareTo()‘) puede generar una carga extra en el rendimiento. Entonces la complejidad teórica es una guía porque la implementación práctica y la distribución de los datos pueden afectar el rendimiento real, haciendo que un algoritmo según la información más eficiente, sea más lento para casos específicos.

6.2. Implementación de Counting Sort

Se implementó el siguiente algoritmo:

```
1   private void countingSort(String attribute) {
2   if (!"quality".equals(attribute)) {
3       collectionsSort(attribute);
4       return;
5   }
6
7   int[] count = new int[101];
8   for (Game game : data) {
9       count[game.getQuality()]++;
10  }
11
12  ArrayList<Game> sorted = new ArrayList<>();
13  for (int i = 0; i <= 100; i++) {
14      for (Game game : data) {
15          if (game.getQuality() == i) {
16              sorted.add(game);
17              count[i]--;
18              if (count[i] == 0) break;
19          }
20      }
21  }
22
23  data.clear();
24  data.addAll(sorted);
```

```
25      }
```

Los resultados que obtuvimos muestran que Counting Sort (para calidad) tiene tiempos de ejecución de 0.873 ms para 10^2, 9.338 ms para 10^4 y 402.415 ms para 10^6 elementos (como se puede observar en el cuadro 5). Si comparamos sus tiempos contra los algoritmos anteriores utilizados para ordenar:

- Para 10^2 elementos, 'Counting Sort' (0.873 ms) es más lento que 'quickSort' (0.117 ms), 'mergeSort' (0.257 ms) e 'insertionSort' (0.483 ms). 'collectionsSort' (0.541 ms) también es más rápido.
- Para 10^4 elementos, 'Counting Sort' (9.338 ms) es más lento que 'quickSort' (3.263 ms) y 'mergeSort' (9.013 ms), y parecido a 'collectionsSort' (9.761 ms).
- Para 10^6 elementos, 'Counting Sort' (402.415 ms) es comparable a 'quickSort' (561.600 ms) y 'mergeSort' (774.036 ms), y mucho más lento que 'collectionsSort' (145.468 ms).

Cuadro 5: Tiempos de ejecución de Counting Sort

Cantidad de Datos	**Tiempo de Ejecución (ms)**
10^2	0,873
10^4	9,338
10^6	402,415

Según la teoría Counting Sort debería ser el más rápido cuando los valores que estamos ordenando van de 0 a 100 pero cuando probamos con un millón de datos, el método que Java usa por defecto para ordenar (Collections.sort) fue más rápido. Tal vez el Counting Sort necesita hacer más trabajo, como crear listas nuevas, lo que lo hace más lento. También puede ser que el método de Java esté mucho mejor hecho que termina siendo el más eficiente. La mayor ventaja de Counting Sort es que puede ser muy rápido porque su tiempo de ejecución es forma lineal, o sea que aumenta de manera según la cantidad de datos. Además, no compara los elementos entre sí como hacen otros métodos, lo que en algunos casos puede hacerlo aún más rápido, siendo aún más como nuestro caso que los valores posibles son pocos y están en un rango limitado (como de 0 a 100).

6.3. Uso de generics en Java para estructuras reutilizables

Java cuenta con la funcionalidad llamada Generics, que permite definir clases, interfaces y métodos con parámetros de tipo como T. En este caso, para que la clase Dataset funcione con cualquier tipo de objeto en lugar de solo Game, se puede parametrizar utilizando generics, declarándola como class Dataset T. El ArrayList también se haría de tipo ArrayList T. Para permitir el ordenamiento y la búsqueda por atributos específicos (precio, categoría, calidad), los objetos de tipo T necesitarían

implementar una interfaz que tenga estos atributos. Además, el uso de interfaces y Comparator Function Objects es fundamental para la comparación y acceso a atributos de tipos genéricos. Usar generics en estructuras de datos, como en la clase Dataset, tiene varias ventajas. Por ejemplo, te permite reutilizar código sin tener que escribir lo mismo una para distintos tipos de objetos. Por ejemplo, se puede usar la misma clase Dataset para manejar más tipos de productos. Otra ventaja es que los generics ayudan a atrapar errores de tipo mientras estás escribiendo el código, antes de que lo ejecutes. También te ahorra tener que hacer casts (cambiar de tipo), porque el compilador ya sabe de qué tipo son los objetos. Esto hace que el código sea mejor de leer. En resumen, usar generics tiene ventajas para hacer trabajos y también hace que tu código sea más claro y fácil de tener.

7. Conclusión

El laboratorio tuvo como objetivo principal analizar algoritmos de ordenamiento y búsqueda aplicándolos sobre listas enlazadas de juegos, utilizando la clase Game para representar los juegos con atributos como nombre, categoría, precio y calidad, y la clase Dataset para realizar las operaciones. Se buscó medir el rendimiento de cada algoritmo en función del tiempo, analizando su comportamiento en la práctica, identificando ventajas y desventajas, y determinando cuál conviene usar en distintos escenarios.

Se comparó la búsqueda lineal y binaria en diferentes métodos del código. El análisis sugiere que, a pesar de la complejidad teórica superior de la búsqueda binaria, su implementación práctica y la distribución específica de los datos (como muchos juegos en la misma categoría, lo que requiere buscar múltiples coincidencias tras encontrar una) pueden hacer que la búsqueda lineal sea más eficiente en ciertos casos, especialmente al comparar muchas cadenas de texto. Adicionalmente, se menciona la implementación de Counting Sort y el uso de generics en Java para crear estructuras reutilizables. El uso de generics permitió reutilizar código, lo cual ayuda a detectar errores de tipo en tiempo de compilación, haciendo el código más claro y fácil de mantener.

En resumen, el laboratorio comparó el rendimiento de algoritmos de ordenamiento y búsqueda en un contexto de gestión de videojuegos, destacando que la eficiencia práctica puede variar según la implementación y la naturaleza de los datos, incluso contrariamente a la complejidad teórica, como se vio en el caso de la búsqueda por categoría.