Escuela de Informática & Telecomunicaciones

Estructuras de Datos & Algoritmos

# Laboratorio n°3

**Autores:**
Allen Mora
Gabriel Varas

**Profesor:**
Marcos Fantóval

**Fecha:**
26/05/2025

# Índice

# 1. Introducción

En este laboratorio se desarrollará un algoritmo de búsqueda de videojuegos según atributos tales como el nombre,categoría,precio y calidad. Esto se logrará mediante el uso de algoritmos de ordenamiento, algoritmos de búsqueda y listas enlazadas.

# 2. Metodología

El funcionamiento general del código se divide en cuatro pasos:

- Generación de datos.
- Ejecución de algoritmos de ordenamiento
- Medición de rendimiento
- Presentación de resultados.

Para completar estos cuatro pasos se crearon métodos que trabajan en conjunto con el fin de lograr ordenar y buscar los videojuegos según el criterio deseado, siendo los posibles criterios: nombre, categoría, precio y calidad.

## 2.1. Métodos

- Game: Es la clase base para cada uno de los videojuegos, en esta se encuentran los atributos de estos mismo con su respectivo constructor además de sus ¨setters¨ y ¨getters¨.

```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;
    }

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

Figura 1: Clase Game

- GenerateData: En esta clase se crean, guardan y cargan los juegos creados de manera aleatoria, esto se logra generando una lista de objetos de la clase ¨Game¨, para cada videojuego se crea un nombre con una combinación aleatoria de dos palabras de un arreglo estático llamado ¨names¨.
  Para la categoría se elige al azar una de otro arreglo estático llamado ¨categories¨.
  El precio se fija como un multiplo de 1.000 entre 1.000 y 7.000. Y como fijan se fija la calidad del juego como un numero entero aleatorio entre 0 y 99.
  Luego estos juegos son guardados en un archivo .csv utilizando el formato name, category, price, quality.
  Para cargar estos juegos lee el archivo .csv y convierte cada linea del archivo en un objeto de la clase Game, y por ultimo en el metodo Main genera 3 archivos .csv llamados : games_100, games_10000 y games_1000000, donde cada uno de esos archivos contiene esa misma cantidad registros de videojuegos.

```java
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;

public class GenerateData {
    static Random random = new Random();
    static String[] names = { "Dragon", "Empire", "Quest", "Galaxy", "Legends", "Warrior" };
    static String[] categories = { "Action", "Adventure", "RPG", "Strategy", "Simulation" };

    private static String generateRandomName() {
        StringBuilder name = new StringBuilder();
        for (int i = 0; i < 2; i++) {
            int index = random.nextInt(names.length);
            name.append(names[index]);
        }
        return name.toString();
    }

    public static ArrayList<Game> generate(int numGames) {
        ArrayList<Game> games = new ArrayList<>();
        for (int i = 0; i < numGames; i++) {
            String name = generateRandomName();
            String category = categories[(int) (Math.random() * categories.length)];
            int price = (random.nextInt(69) + 1) * 1000;
            int quality = random.nextInt(100);
            games.add(new Game(name, category, price, quality));
        }
        return games;
    }

    public static void saveToFile(ArrayList<Game> games, String filename) {
        try (PrintWriter writer = new PrintWriter(new File(filename))) {
            for (Game g : games) {
                writer.println(g.toCSV());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Figura 2: Clase GenerateData 1

```
    public static ArrayList<Game> loadGames(String filename) {
        ArrayList<Game> games = new ArrayList<>();
        try (Scanner sc = new Scanner(new File(filename))) {
            while (sc.hasNextLine()) {
                String[] parts = sc.nextLine().split(",");
                if (parts.length != 4) continue;
                String name = parts[0];
                String category = parts[1];
                int price = Integer.parseInt(parts[2]);
                int quality = Integer.parseInt(parts[3]);
                games.add(new Game(name, category, price, quality));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return games;
    }

    public static void printGames(ArrayList<Game> games) {
        for (Game game : games) {
            System.out.println("Name: " + game.getName() + ", Category: " + game.getCategory() +
                    ", Price: " + game.getPrice() + ", Quality: " + game.getQuality());
        }
    }

    public static void main(String[] args) {
        int[] numGames = {100, 10000, 1000000};
        for (int n : numGames) {
            ArrayList<Game> games = generate(n);
            String filename = "data/games_" + n + ".csv";
            saveToFile(games, filename);
            System.out.println("Generated " + n + " games and saved to " + filename);
        }
    }
}
```

Figura 3: Clase GenerateData 2

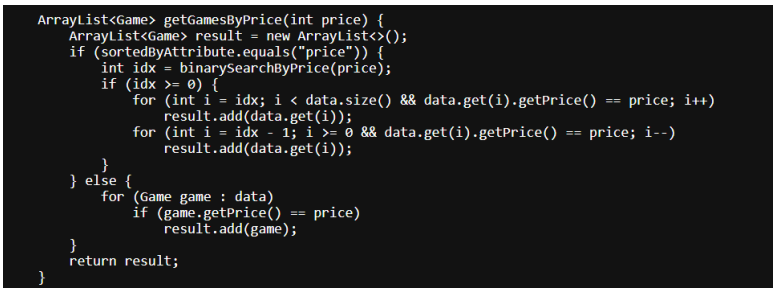
- Dataset: Esta clase una capa intermedia entre los datos sin procesar y los algoritmos necesarios para procesarlos.
  En la variable ¨data¨, se almacena una lista de objetos de la clase ¨Game¨ los cuales son cargados desde los archivos .csv mencionados en el metodo anterior, la variable ¨sortedByAtribute¨ registra el atributo con el cual está ordenado el dataset actualmente.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

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

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

Figura 4: Clase Dataset

Luego dentro de esta se contienen cuatro métodos de búsqueda:

- getGamesByPrice(int price): Este método utiliza búsqueda binaria si los videojuegos estan ordenados por precio, de no ser asi utiliza una busqueda lineal.



```
ArrayList<Game> getGamesByPrice(int price) {
    ArrayList<Game> result = new ArrayList<>();
    if (sortedByAttribute.equals("price")) {
        int idx = binarySearchByPrice(price);
        if (idx >= 0) {
            for (int i = idx; i < data.size() && data.get(i).getPrice() == price; i++)
                result.add(data.get(i));
            for (int i = idx - 1; i >= 0 && data.get(i).getPrice() == price; i--)
                result.add(data.get(i));
        }
    } else {
        for (Game game : data)
            if (game.getPrice() == price)
                result.add(game);
    }
    return result;
}
```

Figura 5: Método getGamesByPrice

- getGamesByPriceRange(int low, int high): Utiliza una búsqueda lineal en todo el dataset.

```
public ArrayList<Game> getGamesByPriceRange(int low, int high) {
    ArrayList<Game> result = new ArrayList<>();
    for (Game game : data) {
        if (game.getPrice() >= low && game.getPrice() <= high)
            result.add(game);
    }
    return result;
}
```

Figura 6: Método getGamesByPriceRange

- getGamesByCategory(string category): Este método entrega una lista de videojuegos que contengan la categoría con la cual se hizo la búsqueda.

```
public ArrayList<Game> getGamesByCategory(String category) {
    ArrayList<Game> result = new ArrayList<>();
    if (sortedByAttribute.equals("category")) {
        for (Game game : data)
            if (game.getCategory().equals(category))
                result.add(game);
    } else {
        for (Game game : data)
            if (game.getCategory().equals(category))
                result.add(game);
    }
    return result;
}
```

Figura 7: Método getGamesByCategory

- getGamesByQuality(string quality): Este método entrega una lista de videojuegos que contengan la calidad buscada.

```
public ArrayList<Game> getGamesByQuality(int quality) {
    ArrayList<Game> result = new ArrayList<>();
    if (sortedByAttribute.equals("quality")) {
        for (Game game : data)
            if (game.getQuality() == quality)
                result.add(game);
    } else {
        for (Game game : data)
            if (game.getQuality() == quality)
                result.add(game);
    }
    return result;
}
```

Figura 8: Método getGamesByQuality

Otros métodos dentro de esta misma función son:

- sortByAlgorithm(string algorithm, string attribute): Este método recibe el nombre del algoritmo de búsqueda y el atributo por el cual se realiza la comparación.

```
public void sortByAlgorithm(String algorithm, String attribute) {
    Comparator<Game> comparator = switch (attribute) {
        case "price" -> Comparator.comparingInt(g -> g.getPrice());
        case "category" -> Comparator.comparing(g -> g.getCategory());
        case "quality" -> Comparator.comparingInt(g -> g.getQuality());
        default -> Comparator.comparingInt(g -> g.getPrice());
    };

    switch (algorithm) {
        case "bubbleSort" -> SortAlgorithms.bubbleSort(data, comparator);
        case "insertionSort" -> SortAlgorithms.insertionSort(data, comparator);
        case "selectionSort" -> SortAlgorithms.selectionSort(data, comparator);
        case "mergeSort" -> data = SortAlgorithms.mergeSort(data, comparator);
        case "quickSort" -> SortAlgorithms.quickSort(data, 0, data.size() - 1, comparator);
        default -> data.sort(comparator);
    }

    sortedByAttribute = attribute;
}
```

Figura 9: Método sortByAlgorithm

- binarySearchByPrice(int price): Realiza una búsqueda binaria sobre el atributo ¨price¨, asumiendo que los videojuegos se encuentran ordenados por este atributo.

```
    private int binarySearchByPrice(int price) {
        int low = 0, high = data.size() - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (data.get(mid).getPrice() == price)
                return mid;
            if (data.get(mid).getPrice() < price)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1;
    }
```

Figura 10: Método binarySearchByPrice

- countingSortByQuality(): Implementa el algoritmo de búsquda ¨Counting Sort¨, crea 101 listas(una para cada valor de calidad posible entre 0 y 100), luego guarda cada videojuego en la lista correspondiente segun su calidad, para luego concatenar todas las listas y reemplazar la lista original.

```
    public void countingSortByQuality() {
        int maxQuality = 100;
        List<List<Game>> buckets = new ArrayList<>(maxQuality + 1);
        for (int i = 0; i <= maxQuality; i++)
            buckets.add(new ArrayList<>());

        for (Game g : data)
            buckets.get(g.getQuality()).add(g);

        ArrayList<Game> sorted = new ArrayList<>();
        for (List<Game> bucket : buckets)
            sorted.addAll(bucket);

        data = sorted;
        sortedByAttribute = "quality";
    }

}
```

Figura 11: Método CountingSortByQuality

- SortAlgorithms: Esta es la clase donde se encuentran los tipos de algoritmos de ordenamiento utilizados durante el laboratorio, contiene cinco métodos, uno para cada algoritmo de ordenamiento diferente, estos métodos son:
    - bubbleSort(): Este método compara dos elementos contiguos y los intercambia en caso de estar en el orden equivocado, realiza este proceso hasta que la lista se encuentre completamente ordenada. Tiene complejidad $O(n^2)$.

```
public class SortAlgorithms {

    public static void bubbleSort(ArrayList<Game> list, Comparator<Game> comparator) {
        int n = list.size();
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (comparator.compare(list.get(j), list.get(j + 1)) > 0)
                    Collections.swap(list, j, j + 1);
    }
```

Figura 12: Método BubbleSort

    - insertionSort(): Este método construye una lista ordenada desde cero insertando los elementos uno por uno en la posición correcta. Tiene complejidad $O(n^2)$.

```
    public static void insertionSort(ArrayList<Game> list, Comparator<Game> comparator) {
        for (int i = 1; i < list.size(); i++) {
            Game key = list.get(i);
            int j = i - 1;
            while (j >= 0 && comparator.compare(list.get(j), key) > 0) {
                list.set(j + 1, list.get(j));
                j--;
            }
            list.set(j + 1, key);
        }
    }
```

Figura 13: Método InsertionSort

    - selectionSort(); Este método encuentra el elemento mínimo para cada iteración y lo coloca en su lugar. Tiene complejidad $O(n^2)$.

```
    public static void selectionSort(ArrayList<Game> list, Comparator<Game> comparator) {
        for (int i = 0; i < list.size() - 1; i++) {
            int min = i;
            for (int j = i + 1; j < list.size(); j++)
                if (comparator.compare(list.get(j), list.get(min)) < 0)
                    min = j;
            Collections.swap(list, i, min);
        }
    }
```

Figura 14: Método SelectionSort

- mergeSort(); Este método dividir la lista en mitades hasta que quede solo un elemento en la lista, luego ordena estas unidades en el orden correcto. Tiene complejidad O(n log n)

```
public static ArrayList<Game> mergeSort(ArrayList<Game> list, Comparator<Game> comparator) {
    if (list.size() <= 1) return list;
    int mid = list.size() / 2;
    ArrayList<Game> left = mergeSort(new ArrayList<>(list.subList(0, mid)), comparator);
    ArrayList<Game> right = mergeSort(new ArrayList<>(list.subList(mid, list.size())), comparator);
    return merge(left, right, comparator);
}

private static 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 15: Método MergeSort

- quickSort(): Este método utiliza un ¨pivote¨ para dividir la lista en elementos menores y mayores. Tiene complejidad O(n log n), pero puede tener complejidad $O(n^2)$ si se elige un pivote incorrecto.

```
    public static void quickSort(ArrayList<Game> list, int low, int high, Comparator<Game> comparator) {
        if (low < high) {
            int pi = partition(list, low, high, comparator);
            quickSort(list, low, pi - 1, comparator);
            quickSort(list, pi + 1, high, comparator);
        }
    }

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

Figura 16: Método QuickSort

- Benchmarck: Esta clase donde se realizan las pruebas de rendimiento de cada uno de los algoritmos anteriormente diseñados.
  Primero se definen los algoritmos de ordenamiento a evaluar y los atributos sobre los cuales se aplicará el ordenamiento, luego se utilizan las siguientes funciones para medir cuanto tiempo demora cada algoritmo:
  - long benchmarkSort(Dataset dataset, String algorithm, String attribute): Este método utiliza System.currentTimeMillis() para calcular cuanto tiempo se demora un ordenamiento en ejecutarse sobre un data set (Existe una función gemela a esta para cada parametro de busqueda).

```
import java.util.ArrayList;

public class Benchmark {
    static String[] sortAlgorithms = {
        "bubbleSort", "insertionSort", "selectionSort",
        "mergeSort", "quickSort", "collectionsSort"
    };
    static String[] attributes = {"price", "category", "quality"};

    public static long benchmarkSort(Dataset dataset, String algorithm, String attribute) {
        long start = System.currentTimeMillis();
        dataset.sortByAlgorithm(algorithm, attribute);
        long end = System.currentTimeMillis();
        return end - start;
    }

    public static long benchmarkSearchPrice(Dataset dataset, int price) {
        long start = System.currentTimeMillis();
        dataset.getGamesByPrice(price);
        long end = System.currentTimeMillis();
        return end - start;
    }

    public static long benchmarkSearchRange(Dataset dataset, int low, int high) {
        long start = System.currentTimeMillis();
        dataset.getGamesByPriceRange(low, high);
        long end = System.currentTimeMillis();
        return end - start;
    }

    public static long benchmarkSearchCategory(Dataset dataset, String category) {
        long start = System.currentTimeMillis();
        dataset.getGamesByCategory(category);
        long end = System.currentTimeMillis();
        return end - start;
    }

    public static long benchmarkSearchQuality(Dataset dataset, int quality) {
        long start = System.currentTimeMillis();
        dataset.getGamesByQuality(quality);
        long end = System.currentTimeMillis();
        return end - start;
    }
```

Figura 17: Método BenchmarkSort

- runSortBenchmarcks(): Este método imprime los resultados por atributo, ademas ejecuta cada algoritmo tres veces para obtener un promedio de tiempo de ejecución, utiliza GenerateData.loadGames() para cargar los datos desde los archivos .csv y además utiliza la clase Dataset para aplicar el ordenamiento.

```
public static void runSortBenchmarks(String datasetPath, String datasetName) {
    System.out.println("====== Benchmark Ordenamiento (" + datasetName + ") ======");
    for (String attribute : attributes) {
        System.out.println("\n>> Atributo: " + attribute);
        System.out.printf("%-15s %-20s\n", "Algoritmo", "Tiempo promedio (ms)");
        for (String algorithm : sortAlgorithms) {
            long totalTime = 0;
            for (int i = 0; i < 3; i++) {
                ArrayList<Game> games = GenerateData.loadGames(datasetPath);
                Dataset ds = new Dataset(games);
                long time = benchmarkSort(ds, algorithm, attribute);
                totalTime += time;
            }
            long avgTime = totalTime / 3;
            String label = (avgTime > 300_000) ? ">300000" : avgTime + "";
            System.out.printf("%-15s %-20s\n", algorithm, label);
        }
    }
}
```

Figura 18: Método runSortBenchmarcks

- runCountingSort(): Este método ejecuta Countingosrt sobre el atributo quality, este algoritmo solo funciona con datos enteros pequeños, por eso no se utiliza con los otros atributos.

```
public static void runCountingSort(String datasetPath, String datasetName) {
    System.out.println("\n====== Counting Sort por Quality (" + datasetName + ") ======");
    long totalTime = 0;
    for (int i = 0; i < 3; i++) {
        ArrayList<Game> games = GenerateData.loadGames(datasetPath);
        Dataset ds = new Dataset(games);
        long start = System.currentTimeMillis();
        ds.countingSortByQuality();
        long end = System.currentTimeMillis();
        totalTime += (end - start);
    }
    long avgTime = totalTime / 3;
    System.out.println("Tiempo promedio: " + avgTime + " ms");
}
```

Figura 19: Método runCountingSort

- runSearchBenchmarks() : Este método ejecuta pruebas de búsqueda sobre un dataset grande, primero realiza una búsqueda lineal sobre un dataset desordenado, luego ordena el dataset por : precio, categoria, y calidad utilizando quickSort y repite las las búsquedas para evaluar la búsqueda binaria.

```
    public static void runSearchBenchmarks(String datasetPath) {
        System.out.println("\n====== Benchmark Búsqueda (solo con dataset grande) ======");
        ArrayList<Game> original = GenerateData.loadGames(datasetPath);
        Dataset ds = new Dataset(new ArrayList<>(original)); // NO ORDENADO

        System.out.println(">> Búsqueda lineal:");
        System.out.println("getGamesByPrice: " + benchmarkSearchPrice(ds, 5000) + " ms");
        System.out.println("getGamesByPriceRange: " + benchmarkSearchRange(ds, 5000, 10000) + " ms");
        System.out.println("getGamesByCategory: " + benchmarkSearchCategory(ds, "RPG") + " ms");
        System.out.println("getGamesByQuality: " + benchmarkSearchQuality(ds, 80) + " ms");

        Dataset dsSorted = new Dataset(new ArrayList<>(original));
        dsSorted.sortByAlgorithm("quickSort", "price");
        System.out.println("\n>> Búsqueda binaria (ordenado por price):");
        System.out.println("getGamesByPrice: " + benchmarkSearchPrice(dsSorted, 5000) + " ms");
        System.out.println("getGamesByPriceRange: " + benchmarkSearchRange(dsSorted, 5000, 10000) + " ms");

        dsSorted.sortByAlgorithm("quickSort", "category");
        System.out.println("getGamesByCategory: " + benchmarkSearchCategory(dsSorted, "RPG") + " ms");

        dsSorted.sortByAlgorithm("quickSort", "quality");
        System.out.println("getGamesByQuality: " + benchmarkSearchQuality(dsSorted, 80) + " ms");
    }
}
```

Figura 20: Método runSearchBenchmarks

- Main: Este es la función principal del programa, primero ejecuta una prueba de rendimiento con los algoritmos de ordenamiento sobre un archivo con 100 videojuegos y luego repite las mismas pruebas, pero solo con el algoritmo ¨Counting Sort¨, luego repite el mismo proceso con archivos de 10.000 y 1.000.000 videojuegos respectivamente.
  Finalmente realiza una búsqueda en un archivo de 1.000.000 videojuegos.

```
public class Main {
    public static void main(String[] args) {
        Benchmark.runSortBenchmarks("data/games_100.csv", "100 elementos");
        Benchmark.runCountingSort("data/games_100.csv", "100 elementos");

        Benchmark.runSortBenchmarks("data/games_10000.csv", "10.000 elementos");
        Benchmark.runCountingSort("data/games_10000.csv", "10.000 elementos");

        Benchmark.runSortBenchmarks("data/games_1000000.csv", "1.000.000 elementos");
        Benchmark.runCountingSort("data/games_1000000.csv", "1.000.000 elementos");

        Benchmark.runSearchBenchmarks("data/games_1000000.csv");
    }
}
```

Figura 21: Clase Main

# 3. Experimentación y Resultados

## 3.1. Generación de Datos

Para evaluar el rendimiento de los algoritmos de ordenamiento y búsqueda, se generaron 3 datasets utilizando la clase GenerateData, cada uno con un numero distintos de juegos.

- 100 Elementos
- 10.000 Elementos
- 1.000.000 Elementos

Cada objeto de la clase Game fue creado aleatoriamente siguiendo las siguientes reglas:

- Nombre: aleatorio combinando dos palabras relacionadas de un arreglo fijo.
- Categoría: seleccionada desde un conjunto de categorías.
- Precio: aleatorio entre 1.000 y 70.000.
- Calidad: aleatorio entre 0 y 100.

Las datos fueron exportados en formato .csv para facilitar su reutilización mas adelante.

## 3.2. Benchmark

Se evaluó el rendimiento de seis algoritmos de ordenamiento para cada uno de los atributos (price, category y quality):

- Bubble Sort.
- Insertion Sort.
- Selection Sort.
- Merge Sort.
- Quick Sort.
- Collections Sort.
- Counting Sort (solo para quality).

Cada algoritmo fue ejecutado tres veces sobre cada dataset y se registró el tiempo promedio en milisegundos.

Cuadro 1: Resultados para ordenamiento por price

| Algoritmo | 100 elementos | 10.000 elementos | 1.000.000 elementos |
|---|---|---|---|
| Bubble Sort | 2 ms | 594 ms | mas de 300.000 ms |
| Insertion Sort | 0 ms | 213 ms | mas de 300.000 ms |
| Selection Sort | 0 ms | 193 ms | mas de 300.000 ms |
| Merge Sort | 1 ms | 11 ms | 708 ms |
| Quick Sort | 0 ms | 14 ms | 57.094 ms |
| Collections Sort | 0 ms | 4 ms | 203 ms |

Cuadro 2: Resultados para ordenamiento por category

| Algoritmo | 100 elementos | 10.000 elementos | 1.000.000 elementos |
|---|---|---|---|
| Bubble Sort | 1 ms | 1256 ms | mas de 300.000 ms |
| Insertion Sort | 0 ms | 360 ms | mas de 300.000 ms |
| Selection Sort | 0 ms | 742 ms | mas de 300.000 ms |
| Merge Sort | 0 ms | 7 ms | 853 ms |
| Quick Sort | 0 ms | 196 ms | mas de 300.000 ms |
| Collections Sort | 0 ms | 4 ms | 189 ms |

Cuadro 3: Resultados para ordenamiento por quality

| Algoritmo | 100 elementos | 10.000 elementos | 1.000.000 elementos |
|---|---|---|---|
| Bubble Sort | 1 ms | 735 ms | mas de 300.000 ms |
| Insertion Sort | 1 ms | 232 ms | mas de 300.000 ms |
| Selection Sort | 0 ms | 196 ms | mas de 300.000 ms |
| Merge Sort | 0 ms | 4 ms | 709 ms |
| Quick Sort | 0 ms | 4 ms | 44.124 ms |
| Collections Sort | 0 ms | 2 ms | 194 ms |

Cuadro 4: Couting Sort

| Dataset | Tiempo promedio (ms) |
|---|---|
| 100 elementos | 0 ms |
| 10.000 elementos | 1 ms |
| 1.000.000 elementos | 59 ms |

## 3.3. Benchmark de Búsqueda

| Método | Algoritmo | Tiempo (ms) |
|---|---|---|
| getGamesByPrice | Lineal | 36 ms |
| getGamesByPrice | Binaria | 1 ms |
| getGamesByPriceRange | Lineal | 27 ms |
| getGamesByPriceRange | Binaria | 9 ms |
| getGamesByCategory | Lineal | 31 ms |
| getGamesByCategory | Binaria | 25 ms |
| getGamesByQuality | Lineal | 22 ms |
| getGamesByQuality | Binaria | 1 ms |

# 4. Conclusiones

En conclusión Merge Sort y Quick Sort ofrecen un mejor rendimiento a medida que el tamaño del dataset aumenta, en contra parte Bubble sort, Insertion Sort y Selection Sort mostraron una baja de rendimiento en cuanto aumentaba la cantidad de datos aumentaba.
En cuanto a la búsqueda se demostró que el utilizar algoritmos de búsqueda binaria sobre listas de datos ordenadas resulta ser mucho más eficiente que la búsqueda lineal. Además Counting Sort para el atributo de calidad demostró ser altamente eficiente con listas con pocos datos.
Por otro lado los resultados obtenidos en las pruebas de rendimiento muestran la importancia de elegir un algoritmo adecuado según las características del dataset con el que se este trabajando.

https://github.com/gabovrs/lab3eda/