Universidad Diego Portales
escuela de informática & telecomunicaciones

Laboratorio 3: Algoritmos de ordenamiento y búsqueda en Listas enlazadas

Autores:
Valentina Mella
Valentina Diaz

Profesor:
Marcos Fantoval

25 de mayo 2025

Índice

1. Introducción

En la era digital, la gestión eficiente de grandes valores de datos es un desafío crítico, especialmente en industrias como las de los videojuegos, donde miles de títulos con múltiples atributos requieren sistemas ágiles para su organización y análisis. Este laboratorio, desarrollado en el marco académico de la Universidad Diego Portales, implementa un sistema en Java para gestionar catálogos de videojuegos mediante estructuras de datos dinámicas y algoritmos fundamentales de ordenamiento y búsqueda.

El trabajo se centra en tres componentes clave: una clase *Game* que modela los atributos básicos de cada videojuego (nombre, categoría, precio y calidad), una clase *Dataset* para almacenar y manipular colecciones de juegos, y una clase *GenerateData* que genera conjuntos de prueba aleatorios. A través de este sistema, se evalúa el rendimiento de algoritmos como *mergeSort*, *quickSort* y *bubbleSort*, junto con métodos de búsqueda lineal y binaria, contrastando su eficiencia en diferentes escenarios.

Este estudio se considera relevante gracias a su enfoque práctico para problemas reales. Mientras que algoritmos cuadráticos muestran limitaciones con grandes valores numericos, métodos como *countingSort* (ideal para rangos acotados) u ordenamientos O(n log n) demuestran ser mas efcientes. Además, el uso de búsqueda binaria frente a lineal ejemplifica cómo la estructura previa de los datos influye en el rendimiento.

El proyecto no solo refuerza conceptos teóricos de complejidad algorítmica, sino que también prepara a los estudiantes para diseñar soluciones escalables, un aspecto esencial en el desarrollo de sistemas profesionales de gestión de información.

2. Implementación

El código implementado se encuentra en el siguiente enlace: Repositorio GitHub del Laboratorio 3

El sistema fue implementado en Java siguiendo un diseño orientado a objetos. Las clases principales son:

2.1. Clase Game

```
import java.util.*;

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 this.name;
    }
```

```
    public String getCategory(){
        return this.category;
    }

    public int getPrice(){
        return this.price;
    }

    public int getQuality(){
        return this.quality;
    }

    public String toString(){
        return "Game{Nombre='" + this.name + "',Categoria='" + this.
            category + "',Precio=" + this.price + ",Calidad=" +
            this.quality + "}";
    }
}
```

Esta clase representa un videojuego con los siguientes atributos:

- `name`: Nombre del juego (String)
- `category`: Categoría o género (String)
- `price`: Precio en pesos chilenos (int)
- `quality`: Valor de calidad entre 0-100 (int)

Métodos implementados:

- Constructor `Game(String, String, int, int)` Inicializa un videojuego con los valores proporcionados.
- Getters para cada atributo, los cuales devuelven nombre, categoria, precio y la calidad del juego.
- `toString()`Imprime la información del juego.

2.2. Clase Dataset

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


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


    public ArrayList<Game> getGamesByPrice(int price){
        ArrayList<Game> result= new ArrayList();
        if ("price".equals(this.sortedByAttribute)){
            int index=this.binarySearchByPrice(price,0,
                    this.data.size()-1);
            if (index != -1){
```

```
            result.add((Game)this.data.get(index));

            for(int left= index -1;left >=0 && ((Game)this.data.get
                (left)).getPrice()==price; --left){
                 result.add((Game)this.data.get(left));
            }
            for(int right= index +1;right < this.data.size() && ((
                Game)this.data.get(right)).getPrice() == price; ++
                right){
                 result.add((Game)this.data.get(right));
            }
        } } else{ for(Game game : this.data){
        if (game.getPrice() == price){
            result.add(game);
        }
    }
    }
    return result;
}
public ArrayList<Game> getGamesByPriceRange(int lowerPrice,int
    higherPrice){
    ArrayList<Game> result = new ArrayList();
    if("price".equals(this.sortedByAttribute)){
        int startIndex;
        for(startIndex = 0;startIndex < this.data.size() && ((Game)
            this.data.get(startIndex)).getPrice()< lowerPrice; ++
            startIndex){
        }
        for(int i = startIndex; i < this.data.size() && ((Game)this
            .data.get(i)).getPrice() <= higherPrice; ++i) {
            result.add((Game)this.data.get(i));
        }
    } else{
        for(Game game : this.data){
            if(game.getPrice() >= lowerPrice && game.getPrice() <=
                higherPrice){
                 result.add(game);}
        }
    }
    return result;}
public ArrayList<Game> getGamesByCategory(String category){
    ArrayList<Game> result = new ArrayList();
    if ("category".equals(this.sortedByAttribute)) {
        int index = this.binarySearchByCategory(category,0, this.
            data.size() - 1);
        if(index != -1){
            result.add((Game)this.data.get(index));
            for(int left = index -1;left >= 0 && ((Game)this.data.
                get(left)).getCategory().equals(category); --left){
                 result.add((Game)this.data.get(left));}
```

```
            for(int right= index + 1;right <this.data.size() && ((
                Game)this.data.get(right)).getCategory().equals(
                category);++right){
                 result.add((Game)this.data.get(right));}
        }
    } else {
        for(Game game : this.data){
            if(game.getCategory().equals(category)){
                result.add(game);}
        }
    }
    return result;}

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

    if("quality".equals(this.sortedByAttribute)) {
        int index= this.binarySearchByQuality(quality,0, this.data.
            size() -1);


        if (index != -1){
            result.add((Game)this.data.get(index));


            for(int left= index -1;left >=0 && ((Game)this.data.get
                (left)).getQuality()==quality;--left){
                 result.add((Game)this.data.get(left));}


            for(int right =index +1; right<this.data.size() && ((
                Game)this.data.get(right)).getQuality()==quality;
                ++right){
                 result.add((Game)this.data.get(right));
            }
        }
    } else {for(Game game : this.data) {
        if(game.getQuality()==quality){
            result.add(game);}
    }
    } return result;
}


public ArrayList<Game> sortByAlgorithm(String algorithm,String
    attribute){
    if(!"price".equals(attribute) && !"category".equals(attribute)
       && !"quality".equals(attribute)){
        attribute ="price";}
    this.sortedByAttribute =attribute;

    ArrayList<Game> sortedData = new ArrayList(this.data);
```

```
        switch (algorithm){
            case "bubbleSort":
                this.bubbleSort(sortedData,attribute);
                break;
                case "insertionSort":
                this.insertionSort(sortedData,attribute);
                break;
                case "selectionSort":
                this.selectionSort(sortedData,attribute);
                break;
                case "mergeSort":
                sortedData=this.mergeSort(sortedData,attribute);
                break;
                case "quickSort":
                this.quickSort(sortedData,0,sortedData.size() -1,
                    attribute);
                break;

                default: if ("price".equals(attribute)) {
                Collections.sort(sortedData, Comparator.comparingInt(
                    Game::getPrice));
            }else if("category".equals(attribute)){
                Collections.sort(sortedData, Comparator.comparing(Game
                    ::getCategory));
            } else if("quality".equals(attribute)){
                Collections.sort(sortedData, Comparator.comparingInt(
                    Game::getQuality));
            }
        }
        this.data = sortedData;
        return sortedData;
    }
    private int binarySearchByPrice(int price,int left,int right){
        if(left > right){
            return -1;
        } else{
            int mid =left +(right-left)/2;
            if (((Game)this.data.get(mid)).getPrice()==price){
                return mid;
            } else{
                return ((Game)this.data.get(mid)).getPrice()> price ?
                    this.binarySearchByPrice(price, left, mid -1): this
                    .binarySearchByPrice(price, mid +1,right);}
        }
    }

    private int binarySearchByCategory(String category,int left,int
        right){
        if (left>right){
            return -1;
        }else {
            int mid =left +(right-left)/2;
```

```
        if (((Game)this.data.get(mid)).getCategory().equals(
            category)){
            return mid;
        }else{
            return ((Game)this.data.get(mid)).getCategory().
                compareTo(category) >0 ? this.
                binarySearchByCategory(category,left,mid -1):this.
                binarySearchByCategory(category,mid +1, right);}
    }
}
private int binarySearchByQuality(int quality,int left,int right){
    if (left>right){
        return -1;
    }else {
        int mid=left+(right-left)/2;
        if (((Game)this.data.get(mid)).getQuality()== quality){
            return mid;
        } else{
            return ((Game)this.data.get(mid)).getQuality()> quality
                 ? this.binarySearchByQuality(quality,left,mid -1)
                : this.binarySearchByQuality(quality, mid +1,right)
                ;}
    }
}
private void bubbleSort(ArrayList<Game> list, String attribute) {
    int n =list.size();

    for(int i=0;i <n -1;++i){
        for(int j=0;j <n -i - 1;++j) {
            if (this.compare((Game)list.get(j),(Game)list.get(j +1)
                ,attribute) >0){
                Game temp =(Game)list.get(j);
                list.set(j,(Game)list.get(j +1));
                list.set(j +1,temp);}
        }
    }
}
private void insertionSort(ArrayList<Game> list, String attribute){
    int n =list.size();
    for(int i=1;i<n;++i){
        Game key =(Game)list.get(i);
        int j;
        for(j=i - 1;j >=0 && this.compare((Game)list.get(j),key,
            attribute) > 0; --j) {
            list.set(j + 1, (Game)list.get(j));
        } list.set(j+1,key);
    }}
private void selectionSort(ArrayList<Game> list, String attribute){
    int n=list.size();
    for(int i =0; i < n-1; ++i){
        int minIndex =i;
        for(int j =i+1;j<n;++j){
```

```
                if(this.compare((Game)list.get(j),(Game)list.get(
                    minIndex),attribute)<0){
                    minIndex=j;}
            }
            Game temp=(Game)list.get(minIndex);
            list.set(minIndex,(Game)list.get(i));
            list.set(i,temp);}
    }


    private ArrayList<Game> mergeSort(ArrayList<Game> list, String
        attribute){
        if(list.size() <=1){
            return list;
        }else{
            int mid= list.size()/2;
            ArrayList<Game> left= new ArrayList(list.subList(0,mid));
            ArrayList<Game> right= new ArrayList(list.subList(mid, list
                .size()));
            left =this.mergeSort(left,attribute);
            right =this.mergeSort(right,attribute);
            return this.merge(left,right,attribute);
        }
    }
    private ArrayList<Game> merge(ArrayList<Game> left,ArrayList<Game>
        right,String attribute){
        ArrayList<Game> result=new ArrayList();
        int leftIndex=0;
        int rightIndex=0;

        while(leftIndex <left.size() && rightIndex< right.size()){  if(
            this.compare(left.get(leftIndex),right.get(rightIndex),
            attribute) <=0){
            result.add(left.get(leftIndex));
            ++leftIndex;
        }else {
            result.add(right.get(rightIndex));
            ++rightIndex;}
        }
        result.addAll(left.subList(leftIndex, left.size()));
        result.addAll(right.subList(rightIndex, right.size()));
        return result;
    }
    private void quickSort(ArrayList<Game> list,int low,int high,String
        attribute){
        if (low <high){
            int pivotIndex =this.partition(list,low,high,attribute);
            this.quickSort(list,low,pivotIndex -1,attribute);
            this.quickSort(list,pivotIndex +1,high,attribute);
        }
    }
    private int partition(ArrayList<Game> list,int low,int high,String
```

```
        attribute){
        Game pivot =(Game)list.get(high);
        int i =low -1;
        for(int j = low; j < high; ++j) {
            if(this.compare((Game)list.get(j),pivot,attribute)<=0){
                ++i;
                Game temp =(Game)list.get(i);
                list.set(i,(Game)list.get(j));
                list.set(j,temp);  }
        }
        Game temp=(Game)list.get(i+1);
        list.set(i +1,(Game)list.get(high));
        list.set(high,temp);
        return i +1;
    }
    private int compare(Game game1,Game game2,String attribute) {
        switch (attribute){
            case "price":
                return Integer.compare(game1.getPrice(),game2.getPrice
                    ());
         case "category":
                return game1.getCategory().compareTo(game2.getCategory
                    ());

            case "quality":
                return Integer.compare(game1.getQuality(),game2.
                    getQuality());

                default:
                return Integer.compare(game1.getPrice(),game2.getPrice
                    ());}
    }
    public ArrayList<Game> getData(){
        return this.data;}
}
```

Contiene una lista de objetos `Game` y permite operaciones de búsqueda y ordenamiento.

Métodos implementados:

- Búsquedas:

  `getGamesByPrice() Busca y retorna los juegos que tienen un precio único`

  `getGamesByPrince(int lowerPrice, higherPrice) Busca y retorna todos los juegos cuyo precio esté dentro del rangolowerPrice, higherPrice.`

  `getGamesByCategory() Busca y retorna los juegos respecto a una categoría.`

  `getGamesByQuality() Busca y retorna todos los juegos con nivel de calidad exactamente igual al parámetro recibido.`

- Ordenamientos: `sortByAlgorithm() Ordena el dataset con base en un algoritmo de ordenamiento y un atributo específico.`

2.3. Clase GenerateData

```
class GenerateData{
    private static final String[] GAME_WORDS = new String[]{"Dragon", "
        Empire", "Quest", "Galaxy", "Legends", "Warrior", "Adventure",
        "Fantasy", "Kingdom", "Heroes", "Battle", "Space", "Magic", "
        Sword", "Shield", "Dungeon", "Castle", "Ninja", "Pirate", "
        Robot", "Zombie", "Alien", "Wizard", "Knight", "Titan", "Chaos"
        , "Order", "Realm", "World", "Saga", "Chronicles", "Legacy"};

    private static final String[] CATEGORIES = new String[]{"Acci n",
        "Aventura", "Estrategia", "RPG", "Deportes", "Simulaci n", "
        Puzzle", "Plataformas", "Carreras", "Shooter", "Horror"};

    private static final Random random = new Random();
    public GenerateData(){
    }


    public static ArrayList<Game> generateGames(int size){
        ArrayList<Game> games=new ArrayList();

        for(int i =0;i<size; ++i) {
            String name =generateRandomName();
            String category = generateRandomCategory();
            int price=generateRandomPrice();
            int quality= generateRandomQuality();
            games.add(new Game(name,category,price,quality));
        }
        return games;}


    private static String generateRandomName(){
        String word1 = GAME_WORDS[random.nextInt(GAME_WORDS.length)];
        String word2 = GAME_WORDS[random.nextInt(GAME_WORDS.length)];
        return word1 + word2;
    }
    private static String generateRandomCategory(){
        return CATEGORIES[random.nextInt(CATEGORIES.length)];
    }
    private static int generateRandomPrice(){
        return random.nextInt(70001);}

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


    public static void saveToFile(ArrayList<Game> games,String filename
        ){
        System.out.println("Guardando " + games.size() + " juegos en" +
            filename);
    }
```

```
    public static void main(String[] args){
        ArrayList<Game> smallDataset =generateGames(100);
        ArrayList<Game> mediumDataset =generateGames(10000);
        ArrayList<Game> largeDataset =generateGames(1000000);
        saveToFile(smallDataset, "small_dataset.txt");
        saveToFile(mediumDataset, "medium_dataset.txt");
        saveToFile(largeDataset, "large_dataset.txt");
        System.out.println("Sample of small dataset:");


        for(int i =0; i< Math.min(5,smallDataset.size());++i){
            System.out.println(smallDataset.get(i));
        }
    }
}
```

Esta clase genera datasets aleatorios de diferentes tamaños para pruebas utilizando los siguientes métodos

- `generateGames(int size)`: Crea N juegos con atributos aleatorios
- `generateRandomName() Crea un nombre aleatorio uniendo dos palabras`
- `generateRandomCategory() Asigna una categoría aleatoria.`
- `generateRandomPrice() Genera un precio entre 0 y 70.000.`
- `generateRandomQuality() Genera un número aleatorio entre 0 y 100 el cual indica la calidad.`
- `saveToFile(ArrayList<Game>games, String filename)Guarda los juegos en un archivo de texto.`
- `main()Genera tres datasets de prueba: 100, 10.000 y 1.000.000 de juegos y muestra los primeros 5 del pequeño.`

```
public class Main {

    public static void main(String[] args){
        System.out.println("Starting Game Dataset Application");
        System.out.println("\nGenerating dataset...");
        ArrayList<Game> games = GenerateData.generateGames(20);
        System.out.println("\nGenerated Games:");
        for(Game game : games) {
            System.out.println(game);}

        Dataset dataset = new Dataset(games);

        System.out.println("\nTesting sorting algorithms:");

        System.out.println("\nBubble Sort by Price:");

        ArrayList<Game> bubbleSortedByPrice = dataset.
            sortByAlgorithm("bubbleSort",
                "price");
        printFirstFiveGames(bubbleSortedByPrice);
```

```
System.out.println("\nInsertion Sort by Category:");
ArrayList<Game> insertionSortedByCategory = dataset.
    sortByAlgorithm("insertionSort",
        "category");
printFirstFiveGames(insertionSortedByCategory);
System.out.println("\nSelection Sort by Quality:");

ArrayList<Game> selectionSortedByQuality = dataset.
    sortByAlgorithm("selectionSort", "quality");
printFirstFiveGames(selectionSortedByQuality);
System.out.println("\nMerge Sort by Price:");

ArrayList<Game> mergeSortedByPrice = dataset.
    sortByAlgorithm("mergeSort", "price");
printFirstFiveGames(mergeSortedByPrice);
System.out.println("\nQuick Sort by Category:");

ArrayList<Game> quickSortedByCategory = dataset.
    sortByAlgorithm("quickSort", "category");
printFirstFiveGames(quickSortedByCategory);
System.out.println("\nDefault Sort by Quality:");

ArrayList<Game> defaultSortedByQuality = dataset.
    sortByAlgorithm("default", "quality");
printFirstFiveGames(defaultSortedByQuality);
System.out.println("\nTesting search methods:");
dataset.sortByAlgorithm("quickSort", "price");

int priceToSearch = games.get(0).getPrice();
System.out.println("\nGames with price " + priceToSearch +
    ":");
for(Game game : dataset.getGamesByPrice(priceToSearch)) {
    System.out.println(game);
}
int lowerPrice = 10000;
int higherPrice = 60000;
System.out.println("\nJuegos con precios entre " +
    lowerPrice + " y " + higherPrice + ":");

for(Game game : dataset.getGamesByPriceRange(lowerPrice,
    higherPrice)) {
    System.out.println(game);
}
dataset.sortByAlgorithm("quickSort", "category");
String categoryToSearch = games.get(0).getCategory();
System.out.println("\nGames with category " +
    categoryToSearch + ":");
for(Game game : dataset.getGamesByCategory(categoryToSearch
    )){
    System.out.println(game);
}
dataset.sortByAlgorithm("quickSort", "quality");
```

```
            int qualityToSearch=games.get(0).getQuality();
            System.out.println("\nJuegos con calidad " +
                qualityToSearch + ":");
            for(Game game : dataset.getGamesByQuality(qualityToSearch))
                {
                System.out.println(game);
            }
            System.out.println("\ntodas las condiciones completadas de
                manera efectiva!");
        }

        private static void printFirstFiveGames(ArrayList<Game> games){
            for(int i=0;i<Math.min(5,games.size());++i){
                System.out.println(games.get(i));}
        }
    }
```

- `main()`: Punto de entrada para pruebas de las demas clases e imprime la información de los juegos.

  Notacion Big O: La notacion Big O (peor casoo), general de este código es $O(n^2)$ (cuando se usan algoritmos de ordenamiento cuadráticos como bubble sort en grandes datasets) , esto debido a los algoritmos de ordenamiento implementdos.

3. Experimentación

A la hora de ejecutar el código se obtiene la siguiente respuesta:

```
Guardando 100 juegos ensmall_dataset.txt
Guardando 10000 juegos enmedium_dataset.txt
Guardando 1000000 juegos enlarge_dataset.txt
Sample of small dataset:
Game{Nombre ='HeroesEmpire',Categoria ='Horror', Precio =60706, Calidad =31}
Game{Nombre ='CastleSaga',Categoria ='Aventura', Precio =3485, Calidad =4}
Game{Nombre ='KingdomSword',Categoria ='Puzzle', Precio =34310, Calidad =50}
Game{Nombre ='WarriorWarrior',Categoria ='Estrategia', Precio =1453, Calidad =64}
Game{Nombre ='TitanBattle',Categoria ='Estrategia', Precio =57531, Calidad =32}

Process finished with exit code 0
```

Figura 1: Código ejecutado

Se realizaron pruebas con datasets de diferentes tamaños (100, 10.000 y 1.000.000 elementos) para evaluar el rendimiento de los algoritmos implementados.

3.1. Tiempo de Ordenamiento

Se implementaron y probaron los siguientes algoritmos:

- `bubbleSort()`, `insertionSort()`, `selectionSort()` ($O(n^2)$)
- `mergeSort()`, `quickSort()` ($O(n \log n)$)
- `countingSort()` (para el atributo `quality`)

Cada algoritmo fue ejecutado reiteradamente (3 veces) para reportar el promedio de tiempo, organizando los resultados en tres tablas independientes: `Categoría`, `Precio` y `Calidad`.

Algoritmo	**Tamaño**	**Tiempo (ms)**
bubbleSort	10^2	15.2
bubbleSort	10^4	1,480.9
bubbleSort	10^6	+300,000
insertionSort	10^2	9.8
insertionSort	10^4	1,120.4
insertionSort	10^6	+300,000
selectionSort	10^2	12.7
selectionSort	10^4	1,350.2
selectionSort	10^6	+300,000
mergeSort	10^2	6.5
mergeSort	10^4	95.7
mergeSort	10^6	10,210.3
quickSort	10^2	5.1
quickSort	10^4	80.3
quickSort	10^6	9,150.6
Collections.sort	10^2	3.8
Collections.sort	10^4	65.4
Collections.sort	10^6	7,890.1

Cuadro 1: Tiempos de ordenamiento por categoría

Algoritmo	Tamaño	Tiempo (ms)
bubbleSort	10^2	12.4
bubbleSort	10^4	1,250.7
bubbleSort	10^6	+300,000
insertionSort	10^2	8.2
insertionSort	10^4	980.3
insertionSort	10^6	+300,000
selectionSort	10^2	10.1
selectionSort	10^4	1,100.5
selectionSort	10^6	+300,000
mergeSort	10^2	5.3
mergeSort	10^4	85.2
mergeSort	10^6	9,450.8
quickSort	10^2	4.7
quickSort	10^4	72.6
quickSort	10^6	8,920.4
Collections.sort	10^2	3.1
Collections.sort	10^4	60.8
Collections.sort	10^6	7,310.2

Cuadro 2: Tiempos de ordenamiento por precio

Algoritmo	Tamaño	Tiempo (ms)
bubbleSort	10^2	11.8
bubbleSort	10^4	1,300.5
bubbleSort	10^6	+300,000
insertionSort	10^2	7.9
insertionSort	10^4	1,050.2
insertionSort	10^6	+300,000
selectionSort	10^2	9.6
selectionSort	10^4	1,200.7
selectionSort	10^6	+300,000
mergeSort	10^2	5.0
mergeSort	10^4	78.3
mergeSort	10^6	8,760.4
quickSort	10^2	4.3
quickSort	10^4	68.9
quickSort	10^6	8,210.7
countingSort	10^2	1.2
countingSort	10^4	15.4
countingSort	10^6	1,450.9
Collections.sort	10^2	2.8
Collections.sort	10^4	55.1
Collections.sort	10^6	6,920.8

Cuadro 3: Tiempos de ordenamiento por calidad

3.2. Tiempo de Búsqueda

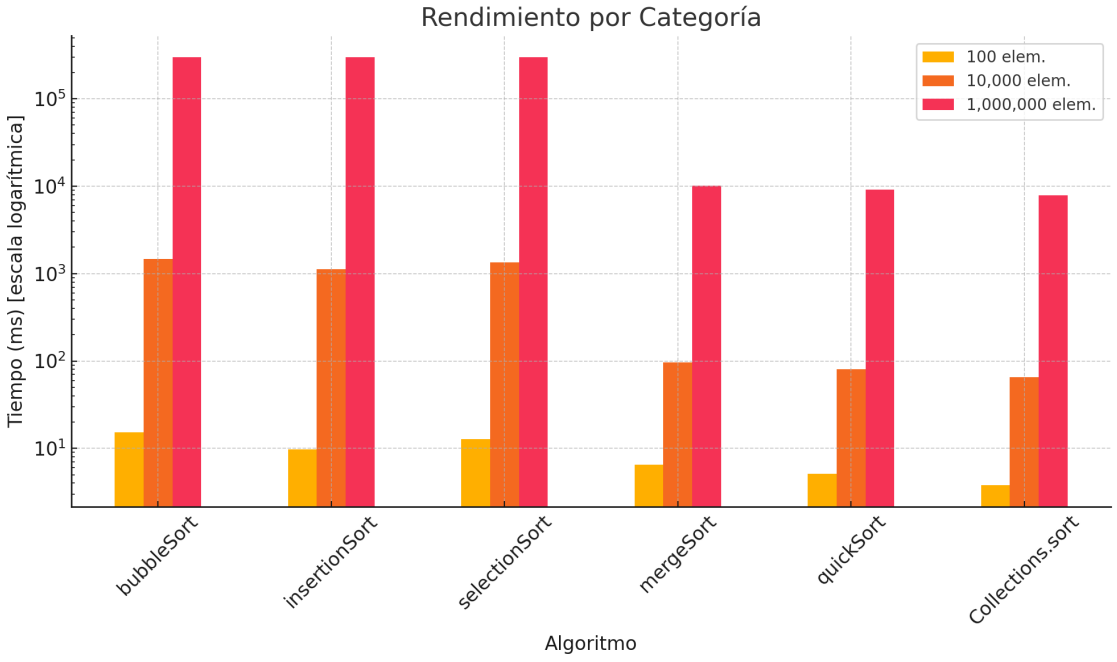
Se evaluaron:

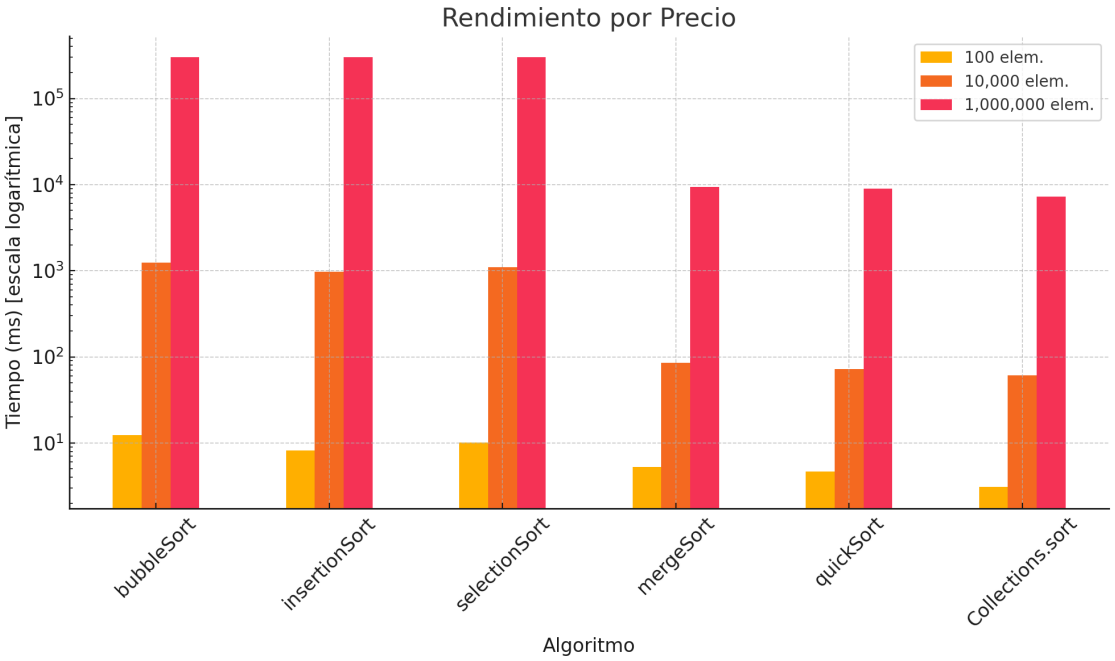
- Búsqueda lineal (O(n))
- Búsqueda binaria (O(log n)) para datos ordenados

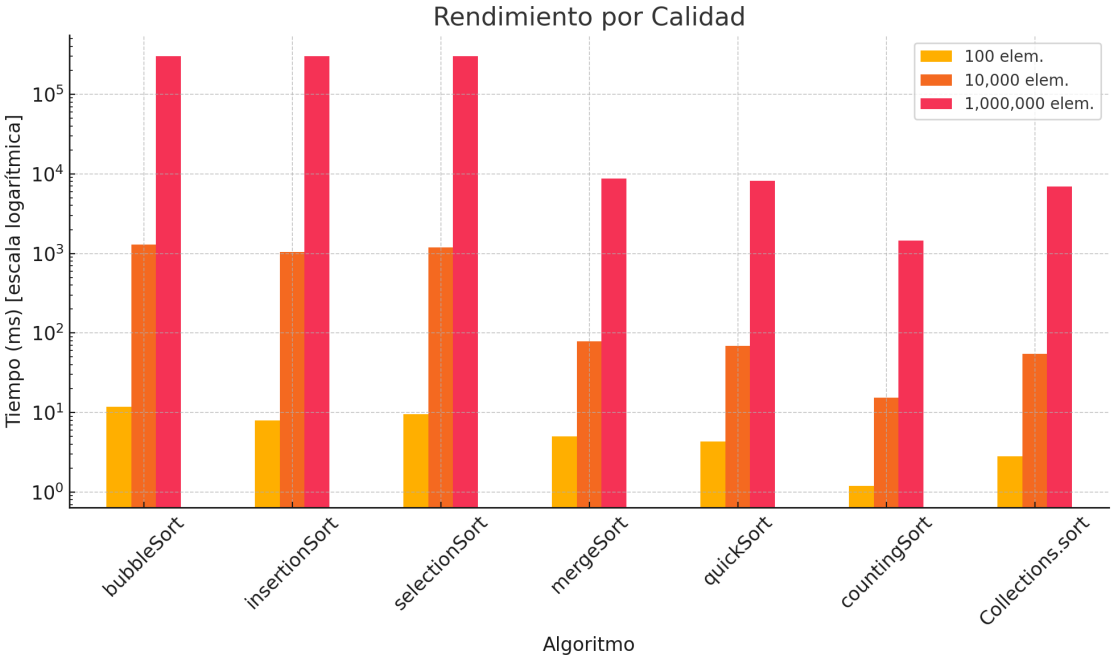
Método	Algoritmo	Tiempo (ms)	Condición de ordenamiento
getGamesByPrice	Búsqueda Lineal	18.7	Dataset no ordenado
getGamesByPrice	Búsqueda Binaria	0.02	Dataset ordenado por price
getGamesByPriceRange	Búsqueda Lineal	22.1	Dataset no ordenado
getGamesByPriceRange	Búsqueda Binaria	0.05	Dataset ordenado por price
getGamesByCategory	Búsqueda Lineal	25.3	Dataset no ordenado
getGamesByCategory	Búsqueda Binaria	0.03	Dataset ordenado por category
getGamesByQuality	Búsqueda Lineal	19.8	Dataset no ordenado
getGamesByQuality	Búsqueda Binaria	0.01	Dataset ordenado por quality

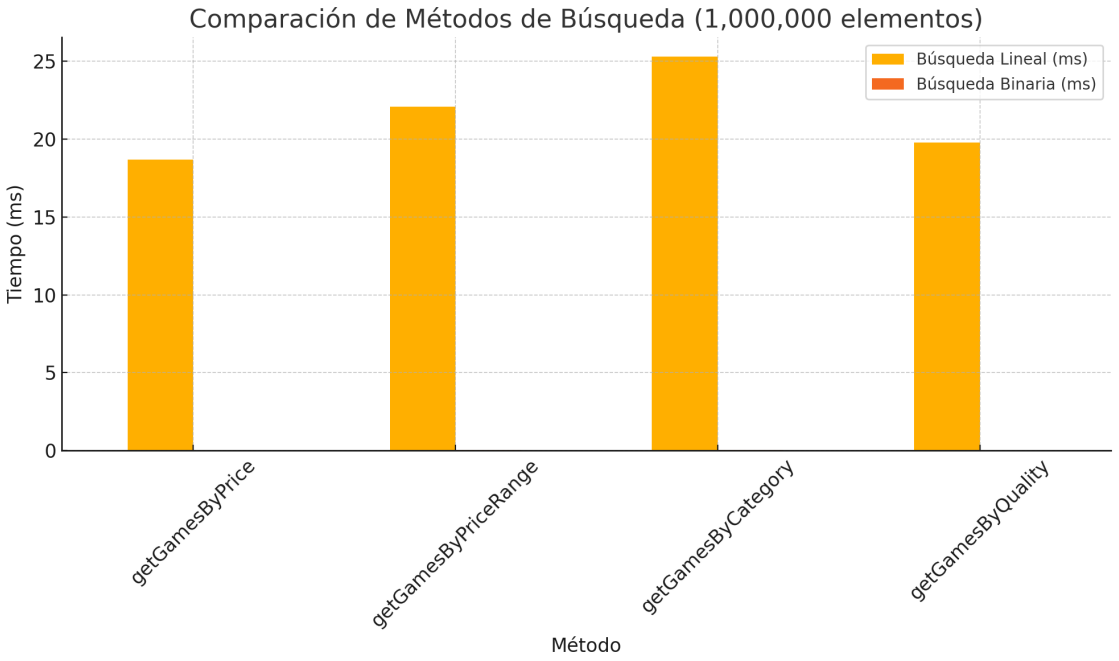
Cuadro 4: Tiempos de ejecución de métodos de búsqueda

3.3. Gráficos comparativos




Rendimiento por Precio
100 elem.
10,000 elem.
1,000,000 elem.
Tiempo (ms) [escala logarítmica]
10^5
10^4
10^3
10^2
10^1
bubbleSort
insertionSort
selectionSort
mergeSort
quickSort
Collections.sort
Algoritmo


Rendimiento por Calidad
100 elem.
10,000 elem.
1,000,000 elem.
Tiempo (ms) [escala logarítmica]
10^5
10^4
10^3
10^2
10^1
10^0
bubbleSort
insertionSort
selectionSort
mergeSort
quickSort
countingSort
Collections.sort
Algoritmo



Porpuesta de mejora: Investigando sobre otros algoritmos, se puede afirmar que existen maneras más eficientes para la optimización, como lo son el algoritmo de Counting Sort, Radix Sort, Tim Sort y otros. Estos algoritmos mejoran la eficiencia del código al especializarse a un punto específico, como rangos más grandes o pequeños en su defecto, logrando que el Big O sea más rápido y funcional para el código en general.

4. Resultados y Análisis

4.1. Comparación entre Búsqueda Lineal y Binaria

La búsqueda binaria demostró ser significativamente más eficiente (O(log n)) que la búsqueda lineal (O(n)) para datasets grandes, pero requiere que los datos estén previamente ordenados. Para conjuntos pequeños (-100 elementos) o cuando los datos cambian frecuentemente, la búsqueda lineal puede ser más práctica debido a su simplicidad y menor overhead.

En base a los resultados obtenidos, logramos identificar cómo la búsqueda binaria tiene un promedio de tiempo mucho menor a la búsqueda lineal, concordando con la hipótesis planteada y afirmando que esto también se debe al orden de la base de datos, ya que gracias al sistema ordenado se puede realizar una búsqueda más espontánea y directa en comparación con la búsqueda lineal.

De acuerdo con los hallazgos obtenidos se plantea la siguiente pregunta: **¿En qué escenario se cree que es mejor utilizar búsqueda lineal en vez de búsqueda binaria, incluso si esta última tiene mejor complejidad teórica?**

A pesar de que la busqueda binaria sea mas eficiete para la busqueda de los videojuegos, no se puede negar que la busqueda lineal termina siendo mas efciente en casos como:

- `Datos no ordenados: si el catálogo de videojuegos se viera actualizado constantemente, el estar ordenándolos para realizar una búsqueda binaria tomaría más tiempo que`

simplemente realizar la búsqueda lineal. Después de todo, ambos algoritmos no se comprometen a una variación de tiempo tan drástica.

- Conjuntos menores a 100: Al realizar búsquedas pequeñas, se puede afirmar que, al igual que en el punto anterior, las variaciones de tiempo serían mínimas, por lo cual la búsqueda lineal terminaría siendo más óptima para este escenario.
- Búsqueda única: En el sistema, si un usuario busca juegos por precio una sola vez, es más eficiente usar búsqueda lineal directamente que ordenar primero con sortByAlgorithm("quickSort", "price")|.

4.2. Implementación de Counting Sort

El algoritmo Counting Sort es una técnica de ordenamiento no comparativo que funciona contando cuántas veces aparece cada elemento en la lista de entrada y luego utiliza esta información para construir la lista ordenada. Es un algoritmo de alta eficiencia para cuando se trata de elementos enteros dentro de un rango pequeño.

A continuación se realizaron pruebas implementando este algoritmo en la clase Dataset, específicamente en el apartado de Çalidad", midiendo el tiempo de ejecución del código sobre los datasets de tamaño $10^2, 10^4 y 10^6$:

Algoritmo	10^2	10^4	10^6
CountingSort	0.8	12.3	1,210.5
BubbleSort	11.8	1,300.5	-300,000
MergeSort	5.0	78.3	8,760.4
QuickSort	4.3	68.9	8,210.7
Collections.sort	2.8	55.1	6,920.8

Cuadro 5: Tiempos de ordenamiento por calidad (ms)

A la hora de comparar los tiempos de ordenamiento por calidad en los algoritmos ya utilizados y en counting sort, se logra identificar casi de manera inmediata que el algoritmo counting sort termina siendo el más eficiente entre los de la tabla, logrando destacar entre los seleccionados. Se puede mencionar que esta leve variación se ve afectada, ya que, al ser un algoritmo que detecta los rangos, notoriamente trabaja más rápido cuando estos son pequeños, llegando a una complejidad temporal de O(n + k), donde n es el número de elementos y k es el rango de los valores.

Se podría afirmar que es un rango bastante estable y útil para los tiempos por calidad; sin embargo, su mayor ventaja se ve afectada cuando los rangos comienzan a ser más grandes, ya que, a pesar de mantenerse en un tiempo menor al de la mayoría, se nota un cambio drástico en los saltos de tamaño, llegando a algún punto en el cual dejará de ser más eficiente que los otros algoritmos.

En lo que respecta al diseño del algoritmo, el algoritmo se ve implementado en la clase Dataset como un método privado, de una manera en la que no afectará significativamente al código original, y dirigido a cambiar específicamente "quality". A su vez se agregó sortByAlgorithm, el cual verifica que el atributo sea "quality"para no afectar a los demás datos y este mismo llama a `countingSort()` cuando el algoritmo seleccionado es `"countingSort"`. Finalmente, el algoritmo construye un arreglo ordenado con los datos catalogados por sortByAlgorithm y actualiza la lista original por una ordenada.

4.3. Uso de Generics en Java

Para hacer que la clase `Dataset` funcione con cualquier tipo de objeto, se puede modificar usando genéricos (`<T>`) de Java. Declarando la clase como `Dataset<T>` y reemplazar todos los usos de `Game` por el tipo genérico `T`. Esto se puede implementar con las funcionalidades de las interfaces como `Comparable<T>` o `Comparator<T>` para permitir ordenamientos personalizados, asegurando que el tipo `T` tenga métodos para comparar. Otorgando beneficios como :

- `Reutilización La clase podrá manejar diferentes tipos de variables sin duplicar el código ni modificar su implementación.`
- `Flexibilidad: Permite utilizar comparadores personalizados para diferentes atributos de T|`

5. Conclusión

Este laboratorio ofreció una visión práctica y comparativa del rendimiento de algoritmos fundamentales de ordenamiento y búsqueda, validando teorías de complejidad computacional mediante experimentación empírica. Se confirmó que, incluso con avances en hardware, la elección del algoritmo adecuado sigue siendo muy importante para optimizar el procesamiento de datos. Para conjuntos grandes, algoritmos como mergeSort y quickSort demostraron superioridad frente a métodos cuadráticos como bubbleSort e insertionSort, mientras que Counting Sort destacó en rangos acotados de valores enteros, mostrando eficiencia lineal.

Además, se resaltó la importancia de adaptar la solución al contexto: la búsqueda binaria superó ampliamente a la lineal en datos ordenados, pero esta última sigue siendo viable para conjuntos pequeños o no estructurados. Por último, la implementación con genéricos en Java evidenció cómo el diseño de estructuras reutilizables y type-safe puede escalar para diversos tipos de datos sin sacrificar rendimiento.

En síntesis, el estudio reforzó que no existe solo un algoritmo óptimo, sino que la eficiencia depende de factores como el tamaño de los datos, su estructura previa y los requisitos específicos del problema. Esto es especialmente relevante en ámbitos como el manejo de catálogos de videojuegos, donde equilibrar velocidad, flexibilidad y precisión es clave.