
Universidad Diego Portales
ESCUELA DE INFORMÁTICA &
TELECOMUNICACIONES


# LAB 3:

# Game Sort


***Autores:***
*Martin Correa*
*Joakin Mac-Auliffe*
*Sergio Pinto*

***Profesor:***
*Marcos Fantoval*


*Link de GitHub*

# Índice

# 1. Introducción

En el presente informe se diseñará e implementará una biblioteca de juegos en Java, la cual tendrá diversas funciones en relación al orden y clasificación de los juegos, implementadas haciendo uso de distintas estructuras de datos y algoritmos de ordenamiento. Una vez implementado y descrito el programa, se realizarán pruebas de estrés mediante la generación de datos de prueba aleatorios, y según los resultados de dichas pruebas se sacarán conclusiones, de esta manera se espera lograr comprender la eficiencia de distintas estructuras de datos y algoritmos de manera experimental.

# 2. Metodología

En esta sección se explicará el funcionamiento del programa, describiendo de forma detallada todas las clases, junto con sus atributos y respectivos métodos.

## 2.1. Clase Game

- **Descripción**

  La clase Game representa un juego con sus atributos, utilizados para identificarlo y posteriormente organizarlos y filtrarlos según los mismos. Los objetos tipo Game son el objeto con el que se trabajará en el programa.

- **Atributos**
  - *String* name: Variable que almacena el nombre del juego
  - *String* category: Variable que almacena la categoría del juego
  - *int* price: Variable que almacena el precio del juego en pesos chilenos
  - *int* quality: Variable que almacena la calidad del juego mediante un entero cuyo valor debe estar entre 0 y 100

- **Métodos**
  - **Game(String name, String category, int price, int quality):** Constructor cuya función es asignar todos sus parámetros a los atributos correspondientes. Además, impone restricciones para los atributos price y quality, siendo estas:
    - El valor del atributo *int price* debe ser mayor o igual a 0
    - El valor del atributo *int quality* debe ser mayor o igual a 0 y menor o igual a 100
  - **Setters y Getters:**
    - int getPrice()

- int getQuality()
- String getName()
- String getCategory()

## 2.2. Clase SearchAndSortMethods

- **Descripción**

  La clase SearchAndSortMethods cumple la función de alojar todos los métodos de búsqueda y ordenamiento que posteriormente serán utilizados en el programa.

- **Atributos**

  La clase SearchAndSortMethods no cuenta con atributos, pues su objetivo es almacenar métodos para mantener más organizado el código.

- **Métodos**

  - **int binarySearch(ArrayList<Game> arrayList, String attribute, int price1, int price2, String category, int quality)):** Método que permite encontrar la posición de un juego que cumpla el parámetro ingresado, realizando la búsqueda de forma binaria.

### 2.2.1. Implementación BinarySearch

Se utiliza una variable llamada *mid* de tipo entero, que se ubica en la parte media del ArrayList, además de otras dos variables enteras que representan ambos extremos (*left*, *right*). Luego se compara el valor en la posición *mid* con el parámetro ingresado. En caso de que coincidan, se retorna la posición de *mid*. Si *mid* es mayor al parámetro, se reubica *right* un lugar por detrás de *mid*, y *mid* se posiciona en la mitad entre *left* y *right*. En caso de que el parámetro sea mayor a *mid*, se realiza el procedimiento anterior invertido, es decir, *left* se reubica por delante de *mid*, y *mid* se posiciona en la mitad entre ambos extremos. Finalmente se vuelve a iniciar el ciclo.

- **void bubbleSort(ArrayList<Game> arrayList, String attribute):** Método que ordena el ArrayList de forma ascendente según el parámetro asignado, empleando el método BubbleSort para ordenar el arreglo.

### 2.2.2. Implementación BubbleSort

Se utiliza un ciclo de tipo *for* para recorrer el ArrayList en su totalidad y de esta manera, asegurar un orden efectivo. Dentro de este ciclo, se implementa un nuevo *for* el cual es el encargado de ordenar el ArrayList.

Su funcionamiento consiste en recorrer cada posición del arreglo, verificando que el elemento siguiente sea mayor o igual al de la posición en que se encuentra. En caso de que esto no se cumpla, intercambia ambas posiciones de los elementos.

- **void insertionSort(ArrayList<Game> arrayList, String attribute):** Método que ordena el ArrayList de forma ascendente según el parámetro asignado, empleando el método InsertionSort para ordenar el arreglo.

#### 2.2.3. Implementación InsertionSort

Se utiliza un ciclo de tipo *for* para recorrer el ArrayList en su totalidad y de esta manera, asegurar un orden efectivo. Dentro de este ciclo, se implementa una variable auxiliar *key* que representa el valor en la posición $i$ del arreglo, el cual, corresponde al elemento que se desea posicionar de forma correcta, además se crea una variable $j$ con valor $i$-1. Posteriormente, se crea un ciclo *while* que se ejecutará tantas veces como $j$ sea mayor o igual 0, y el elemento del arreglo en la posición $j$ sea mayor al parámetro. Dentro de este ciclo, se cambia de posición el valor situado en $j$ a la ubicación $j$+1, esto explicado de forma coloquial se puede determinar cómo 'ir haciendo espacio' para ubicar a *key* en su posición correcta. Una vez que la condición del *while* no se cumple, se posiciona el elemento *key* en el lugar $j$+1 del arreglo.

- **void selectionSort(ArrayList<Game> arrayList, String attribute):** Método que ordena el ArrayList de forma ascendente según el parámetro asignado, empleando el método SelectionSort para ordenar el arreglo.

#### 2.2.4. Implementación SelectionSort

Se utiliza un ciclo de tipo *for* para recorrer el ArrayList en su totalidad y de esta manera, asegurar un orden efectivo. Además, dentro de este ciclo se crea la variable *minimumIndex*, el cual, representa la posición del elemento con menor valor que se encuentre en el arreglo (valor inicial $i$). Posteriormente, se utiliza otro ciclo *for* que compara el valor del elemento en el que se encuentra situado ($j$), con el de la posición $i$, en caso de que el elemento en $j$ sea menor, se modifica el valor de *minimumIndex* por $j$. Una vez finalizado el ciclo, se intercambia el valor de la posición *minimumIndex* por el valor que se encuentra en $i$.

- **void mergeSort(ArrayList<Game> arrayList, String attribute):** Método que ordena el ArrayList de forma ascendente según el parámetro asignado, empleando el método MergeSort para ordenar el arreglo.

### 2.2.5. Implementación MergeSort

Se crea una variable de tipo entero llamada *mid* que representa la posición media de *arrayList*, luego se crea una ArrayList de tipo *Game* llamada *left* que almacena los datos de *arrayList* desde el primer dato hasta *mid*, además se crea una segunda variable del mismo tipo llamada *right* que almacena desde *mid* hasta el último dato. Luego la función se llama de manera recursiva, una vez para *left* y luego para *right*, el objetivo de la recursión es dividir el ArrayList hasta que *left* y *right* sean arreglos con un elemento, para poder compararlos y posicionarlos de manera correcta en el *arrayList* original. Posteriormente, se crean variables de tipo entero inicializadas en 0, las cuales son: $i$, $j$, $k$. Tales variables servirán para posicionar los elementos en orden. A continuación, se utiliza un ciclo *while* que se ejecuta tantas veces como $i$ sea menor al tamaño de *left*, y $j$ sea menor al tamaño de *right*. Dentro del ciclo, se crea la variable *takeleft* de tipo Boolean, el cual su valor dependerá si el contenido en la posición $i$ del arreglo *left* es menor o igual al de la posición $j$ de *right*. Al finalizar el ciclo, se comprueba que *takeleft* sea verdadero, en caso de que se cumpla, el elemento en la posición $i$ del arreglo *left* se guarda en *arrayList* (el arreglo original de esa recursión) en la posición $k$, aumentando el valor de $i$ y $k$ luego de realizarlo. Si no se cumple aquella condición, se realiza el mismo procedimiento, con la diferencia que el elemento a ingresar es el de la posición $j$ en el arreglo *right*. En caso de que en alguno de los dos arreglos haya quedado algún elemento sin ingresar, se entra a un ciclo *while* el cual los registrará con la misma lógica que el explicado anteriormente. Ya que este método funciona de forma recursiva, el procedimiento se repetirá con las llamadas anteriores, hasta finalizar en la función original.

- **void quickSort(ArrayList<Game> arrayList, String attribute):** Método que ordena el ArrayList de forma ascendente según el parámetro asignado, empleando el método QuickSort para ordenar el arreglo.

### 2.2.6. Implementación QuickSort

La lógica con la que funciona QuickSort es: tomar un elemento del arreglo como pivote (suele ser el último), con el objetivo de dividir el arreglo en parte 'derecha' e 'izquierda', donde a la izquierda van los elementos que son menores al pivote, y en el derecho los mayores, además de subdividirse posteriormente de forma 'recursiva'. Sin embargo, en este caso se utilizó un Stack para realizar la operación.

Se crea una variable de tipo entero llamada $n$ que representa el tamaño de *arrayList*, además se crea un Stack *stack* que guarda arreglos de números enteros, dicho arreglo almacenará un par de números que representan las posiciones de los extremos en el rango que se desea ocupar, en este caso

se inicializa desde 0 hasta $n$-1 (límites del arreglo). Luego se crea un ciclo *while* que se ejecutará mientras *stack* no se encuentre vacío. Se utiliza un arreglo *range* que va a ser igual al primer elemento de *stack*, además de las variables int *low*, int *high* que se igualarán a *range[0], range[1]* (limites del subarreglo). A continuación se crea el pivote con el valor del elemento en la posición *high* del subarreglo, además de una variable $i$ inicializada en *low*-1. Se crea un ciclo *for* que recorrerá el subarreglo. Dentro de *for* se inicializa la variable *condition* de tipo Boolean, el cual, será verdadero en caso de que el elemento del subarreglo en la posición $j$ (variable de *for*) sea menor o igual al pivote, en caso contrario, será falso. Si *condition* se cumple, $i$ aumenta su valor en 1 e intercambian de posición los elementos del subarreglo en los lugares $i$ y $j$. Una vez finalizado el ciclo, se intercambia la posición del pivote con el elemento en $i$+1, permitiendo que el pivote se ubique en el lugar correspondiente. Finalmente se crea la variable *pi* con el valor *i+1*, puesto que será utilizado como nuevo límite para los subarreglos posteriores. Dicho esto, se agregan a *stack* dos nuevos subarreglos, uno que va desde *low* hasta *pi*-1 y otro desde *pi*+1 hasta *high*, continuando un nuevo ciclo de ordenamiento.

- **void countingSort(ArrayList<Game> arrayList, String attribute):** Método que ordena el ArrayList de forma ascendente según el parámetro asignado, empleando el método CountingSort para ordenar el arreglo.

### 2.2.7. Implementación CountingSort

Se declaran dos valores enteros llamados *maxQuality* y *minQuality*, cada uno almacenará el valor mínimo y máximo que se encuentre en *arrayList*. Dichos valores se asignan recorriendo el arreglo de forma completa utilizando un ciclo *for*. Luego se crea la variable *range*, cuyo valor corresponde al rango de los valores encontrados (*maxQuality* - *min Quality*) -1, su función es asignar el tamaño que tendrá el nuevo arreglo de enteros llamado *count*, el cual, utilizando un ciclo *for*, servirá para contar cuantas veces se repite cada valor de calidad. Posteriormente, se crea un ciclo *for* para recorrer *count*, con el objetivo de acumular la cantidad de valores que son menores o iguales respecto al valor de la posición $i$. A continuación, se crea un arreglo de objetos *Game* llamado *output*, que servirá para almacenar los elementos ordenados de forma correcta. Luego, se crea un ciclo *for* que recorre *arrayList* de forma inversa. Dentro del ciclo, se crea la variable $q$ que representa la calidad del elemento en *arrayList* en la posición $i$. Utilizando $q$ y el arreglo *count*, se le asigna al elemento $i$ de *arrayList* su posición correspondiente en *output*, que equivale a su lugar en la lista ordenada. Además, se decrementa el valor del elemento posicionado en $q$ - *minQuality*, ya que uno de esos valores ya fue ingresado. Finalmente, se recorre *arrayList*, con el objetivo de copiar elemento por elemento del

arreglo *output*, permitiendo que el arreglo original esté ordenado.

## 2.3. Clase Dataset

- **Descripción**

  Un objeto tipo Dataset representa un arreglo de juegos (objetos de tipo Game), los cuales, pueden ser ordenados o filtrados en base a diversos algoritmos y funciones que posee la clase.

- **Atributos**

  - *ArrayList<Game>* data: Arreglo dinámico que almacena objetos de la clase Game.
  - *String* sortedByAtributte: Variable que almacena el atributo por el que posteriormente será ordenado el arrayList *data*. Los valores que puede tomar son los siguientes:
    - Category
    - Quality
    - Price
  - *SearchAndSortMethods* SearchAndSortMethods: Objeto de tipo searchAndSortMethods cuya función es importar todos los métodos de búsqueda y ordenamiento de dicha clase para utilizarlos en la clase Dataset.

- **Métodos**

  - *ArrayList<Game>* getGamesByPrice(int price): Método que ordena y retorna un arreglo dinámico con todos los juegos cuyos precios sean iguales. Ordena binaria o linealmente dicho arreglo, dependiendo de si la lista ya se encuentra ordenada por precio o no.
  - *ArrayList<Game>* getGamesByPriceRange(int lowerPrice, int higherPrice): Método que pide como parámetro un rango de precios los cuales serán utilizados para ordenar y retornar una arrayList con todos los precios dentro de ese rango. Ordena de manera binaria o linealmente dependiendo de si la lista está ordenada por precio o no.
  - *ArrayList<Game>* getGamesByCategory(String category): Método que ordena y retorna un arrayList con todos los juegos que sean de la misma categoría. Ordena de manera binaria o lineal dependiendo de si la lista ya viene ordenada por categoría o no.
  - *ArrayList<Game>* getGamesByQuality(int quality): Método que ordena y retorna un arrayList con todos los juegos que tengan la misma cate-

goría. Ordena de manera binaria o lineal dependiendo de si la lista está previamente ordenada por calidad o no.

- *void* sortByAlgorithm(String algorithm, String attribute): Método que ordena el atributo *arrayList<Game>* data según el algoritmo ingresado en el primer parámetro. Dicho parámetro puede tomar los siguientes valores:
  - bubbleSort
  - insertionSort
  - selectionSort
  - mergeSort
  - quickSort
  - countingSort

  En caso de que el parámetro *String* algorithm no coincida con ninguno de los valores anteriormente mostrados, utilizará *Collections.sort()* para ordenar el arreglo.

  El parámetro *String* attribute determina en base a qué atributo se va a ordenar el arreglo, dicho parámetro puede tomar los siguientes valores:
  - Category
  - Quality
  - Price

  Este método funciona llamando los métodos del algoritmo de ordenamiento correspondiente mediante el atributo *SearchAndSortMethods* SearchAndSortMethods, declarado al principio de la clase. Cada algoritmo dentro de la clase SearchAndSortMethods requiere como parámetros el arreglo a ordenar, y el atributo por el cual se va a ordenar. Para llamar al método a utilizar, se utiliza el atributo *arrayList<Game>* data y el parámetro *String* attribute (por ejemplo, *SearchAndSortMethods.bubbleSort(data, attribute)*, en caso de que *String* algorithm sea bubbleSort)

- *public static void* main(String[] args): Método cuya función es verificar que el resto de los métodos de la clase funcionen correctamente. Instancia un dataset e ingresa 4 objetos en él de tal manera de utilizarlo con todos los algoritmos. Prueba cada algoritmo de ordenamiento imprimiendo el dataset mediante un ciclo *for* y variando el atributo por el cual se ordena. Posteriormente, crea arreglos dinámicos de objetos tipo *Game* y los asigna a los métodos getGamesByPrice, getGamesByPriceRange,

getGamesByCategory y getGamesByQuality. Finalmente imprime dichos arreglos dinámicos.

## 2.4. Clase GenerateData

- **Descripción**

  La clase GenerateData genera información de cada juego, tal como el nombre, categoría, precio y calidad. Hace esto gracias a la librería random lo cual provoca que se pueda generar atributos de manera aleatoria.

- **Atributos**

  - *String[]* words: Arreglo tipo string que almacena nombres para crear, aleatoriamente, el nombre de uno de los juegos.
  - *String[]* categories: Arreglo tipo string que almacena tipos de categoría que serán asignadas a cierto juego utilizando la librería random.
  - *Random* random: Utilizado para inicializar la librería random.

- **Métodos**

  - *ArrayList<Game> generateData(int n)*: Método que crea $n$ objetos de tipo *Game* y posteriormente le asigna atributos aleatoriamente, mediante la librería Random. Los atributos se crean en base a las siguientes condiciones:
    - El atributo *name* se crea concatentando aleatoriamente dos palabras del atributo *String[]* words, declarado previamente en la clase. Los elementos que conforman dicho atributo son los siguientes: "Dragon", "Empire", "Quest", "Galaxy", "Fant", "Legends", "Warrior", "Pergio", "Tourbes", "Do", "While", "Ohliver", "Sinto", "Super", "Ovalous".
    - El atributo *category* se crea concatenando aleatoriamente dos palabras del atributo *String[]* categories, declarado previamente en la clase. Los elementos que conforman dicho atributo son los siguientes: "Accion", "Aventura", "Estrategia", "RPG", "Deportes", "Simulacion".
    - El atributo *price* se crea generando un número entero aleatorio entre 1000 y 70000.
    - El atributo *quality* se crea generando un número entero aleatorio entre 0 y 100.

    El método crea un arreglo dinámico y mediante un ciclo *for* que itera $n$ veces, llena dicho arreglo dinámico con $n$ cantidad de objetos. Finalmente,

retorna el arreglo dinámico.

- *void* saveGamesToCSV(ArrayList<Game> games, String filename): Método que guarda un arreglo de objetos tipo Game en un archivo .csv, con el objetivo de utilizarlo como muestra de datos para ordenar y así poder medir experimentalmente los tiempos de ejecución de los distintos algoritmos del programa.
- *ArrayList<Game>* readGamesFromCSV(String filename): Método que permite obtener los datos de los archivos .csv creados en el método *saveGamesToCSV* y asignarlos a distintos datasets, con el objetivo de utilizarlos para determinar experimentalmente el tiempo de ejecución de los distintos algoritmos del programa.
- *public static void* main(String[] args): Método cuya función es verificar que el resto de los métodos de la clase funcionen correctamente. Primero, instancia un objeto de tipo *GenerateData* y mediante el mismo crea un arreglo dinámico llamado *games* el cual contiene 20 objetos de tipo *Game* generados aleatoriamente. Posteriormente, imprime dicho arreglo dinámico y lo guarda a un archivo .csv haciendo uso del método *saveGamesToCSV*. Finalmente, utiliza el método *readGamesFromCSV* para leer el archivo .csv previamente guardado y lo vuelve a imprimir.

# 3. Experimentación

## 3.1. Medición del tiempo de ordenamiento

Se midió el tiempo que demoran los algoritmos utilizados en el programa mediante el método *System.currentTimeMillis()*. Los datasets utilizados constan de $10^2$, $10^4$ y $10^6$ objetos tipo Game respectivamente, los cuales fueron generados y almacenados en un archivos .csv gracias a la clase GenerateData. Los resultados obtenidos son los siguientes:

Cuadro 1: Tiempos de ejecución de ordenamiento para el atributo *quality*

| **Algoritmo** | **Tamaño del dataset** | **Tiempo (milisegundos)** |
|---|---|---|
| bubbleSort | $10^2$ | 299ms |
| bubbleSort | $10^4$ | 716ms |
| bubbleSort | $10^6$ | Más de 300 segundos |
| insertionSort | $10^2$ | 357ms |
| insertionSort | $10^4$ | 486ms |
| insertionSort | $10^6$ | Más de 300 segundos |
| selectionSort | $10^2$ | 298ms |
| selectionSort | $10^4$ | 469ms |
| selectionSort | $10^6$ | Más de 300 segundos |
| mergeSort | $10^2$ | 378ms |
| mergeSort | $10^4$ | 360ms |
| mergeSort | $10^6$ | 893ms |
| quickSort | $10^2$ | 339ms |
| quickSort | $10^4$ | 344ms |
| quickSort | $10^6$ | 93524ms |
| Collections.sort | $10^2$ | 336ms |
| Collections.sort | $10^4$ | 311ms |
| Collections.sort | $10^6$ | 469ms |

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

| **Algoritmo** | **Tamaño del dataset** | **Tiempo (milisegundos)** |
|---|---|---|
| bubbleSort | $10^2$ | 304ms |
| bubbleSort | $10^4$ | 622ms |
| bubbleSort | $10^6$ | Más de 300 segundos |
| insertionSort | $10^2$ | 321ms |
| insertionSort | $10^4$ | 402ms |
| insertionSort | $10^6$ | Más de 300 segundos |
| selectionSort | $10^2$ | 298ms |
| selectionSort | $10^4$ | 474ms |
| selectionSort | $10^6$ | Más de 300 segundos |
| mergeSort | $10^2$ | 336ms |
| mergeSort | $10^4$ | 414ms |
| mergeSort | $10^6$ | 1090ms |
| quickSort | $10^2$ | 316ms |
| quickSort | $10^4$ | 360ms |
| quickSort | $10^6$ | 1081ms |
| Collections.sort | $10^2$ | 324ms |
| Collections.sort | $10^4$ | 461ms |
| Collections.sort | $10^6$ | 758ms |

Cuadro 3: Tiempos de ejecución de ordenamiento para el atributo *category*

| **Algoritmo** | **Tamaño del dataset** | **Tiempo (milisegundos)** |
|---|---|---|
| bubbleSort | $10^2$ | 317ms |
| bubbleSort | $10^4$ | 1029ms |
| bubbleSort | $10^6$ | Más de 300 segundos |
| insertionSort | $10^2$ | 351ms |
| insertionSort | $10^4$ | 513ms |
| insertionSort | $10^6$ | Más de 300 segundos |
| selectionSort | $10^2$ | 309ms |
| selectionSort | $10^4$ | 736ms |
| selectionSort | $10^6$ | Más de 300 segundos |
| mergeSort | $10^2$ | 359ms |
| mergeSort | $10^4$ | 319ms |
| mergeSort | $10^6$ | 1047ms |
| quickSort | $10^2$ | 316ms |
| quickSort | $10^4$ | 553ms |
| quickSort | $10^6$ | Más de 300 segundos |
| Collections.sort | $10^2$ | 302ms |
| Collections.sort | $10^4$ | 333ms |
| Collections.sort | $10^6$ | 536ms |

## 3.2. Tiempos de ejecución de búsqueda

Para calcular los tiempos de ejecución de los métodos que utilizan binarySearch se debe calcular el tiempo que demora ordenar los datos. Para ello, se utiliza Collections.sort() y se mide el tiempo que tarda en ejecutarse. Una vez obtenido este valor, se le restará al tiempo que demora en ejecutrase el método binarySearch correspondiente. De esta manera, se obtiene el tiempo que demora en ejecutarse el algoritmo binarySearch. Los parámetros utilizados para calcular los tiempos son:

- getGamesByPrice(10000)
- getGamesByPriceRange(10000, 50000)
- getGamesByCategory("RPG")
- getGamesByCategory(50)

Cuadro 4: Tiempos de ejecución por método y algoritmo de búsqueda

| **Método** | **Algoritmo** | **Tiempo (segundos)** |
|---|---|---|
| getGamesByPrice | linearSearch | 374ms |
| getGamesByPrice | binarySearch | 343ms |
| getGamesByPriceRange | linearSearch | Más de 300 segundos |
| getGamesByPriceRange | binarySearch | Más de 300 segundos |
| getGamesByCategory | linearSearch | 405ms |
| getGamesByCategory | binarySearch | 369ms |
| getGamesByQuality | linearSearch | 360ms |
| getGamesByQuality | binarySearch | 357ms |

## 3.3. Gráficos

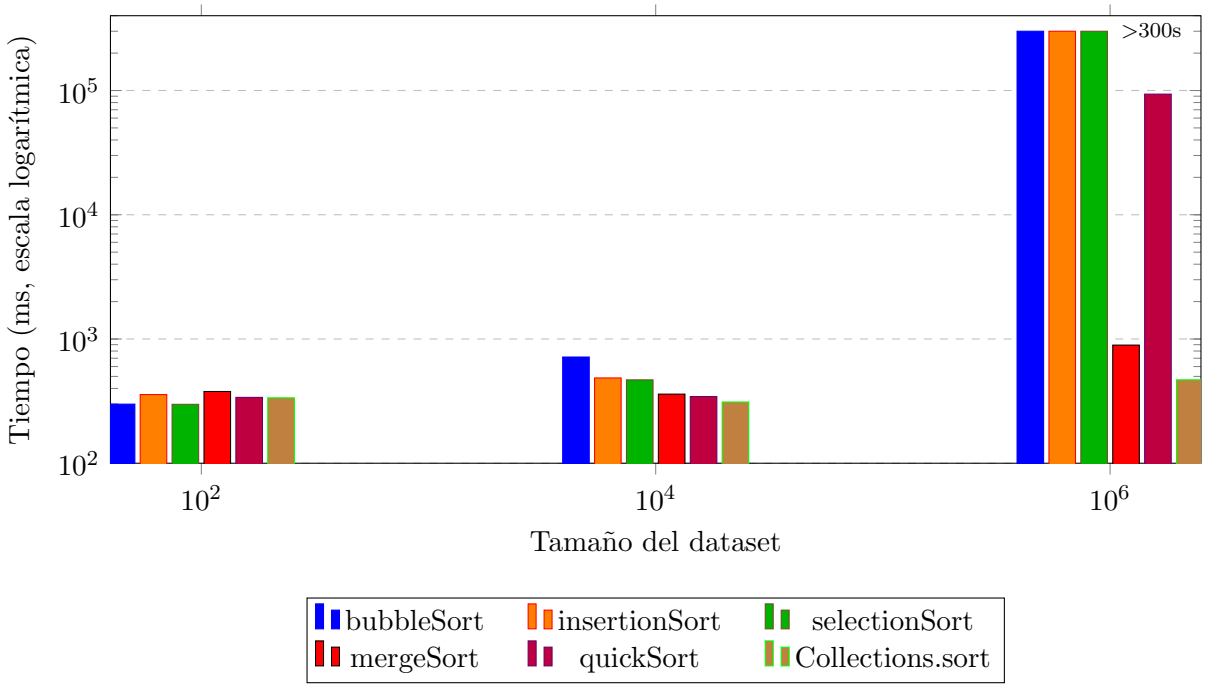


Figura 1: Comparación logarítmica de tiempos de ejecución para diferentes algoritmos de ordenamiento y tamaños de dataset al ordenar según el atributo *quality*

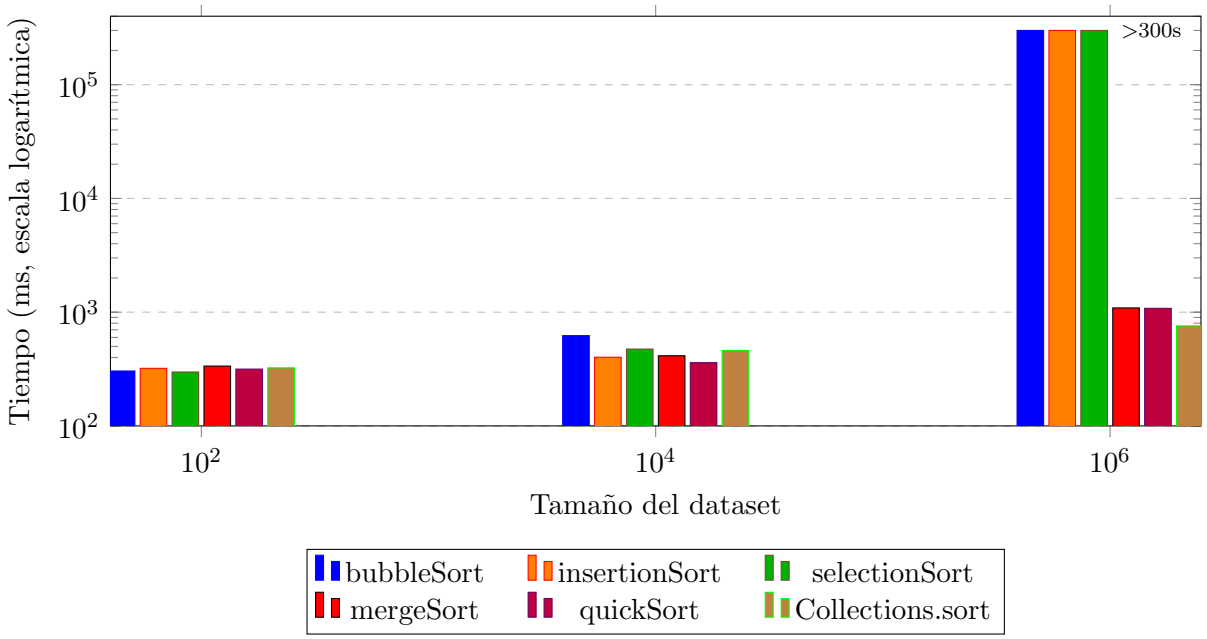


Figura 2: Comparación logarítmica de tiempos de ejecución para diferentes algoritmos de ordenamiento y tamaños de dataset al ordenar según el atributo *price*

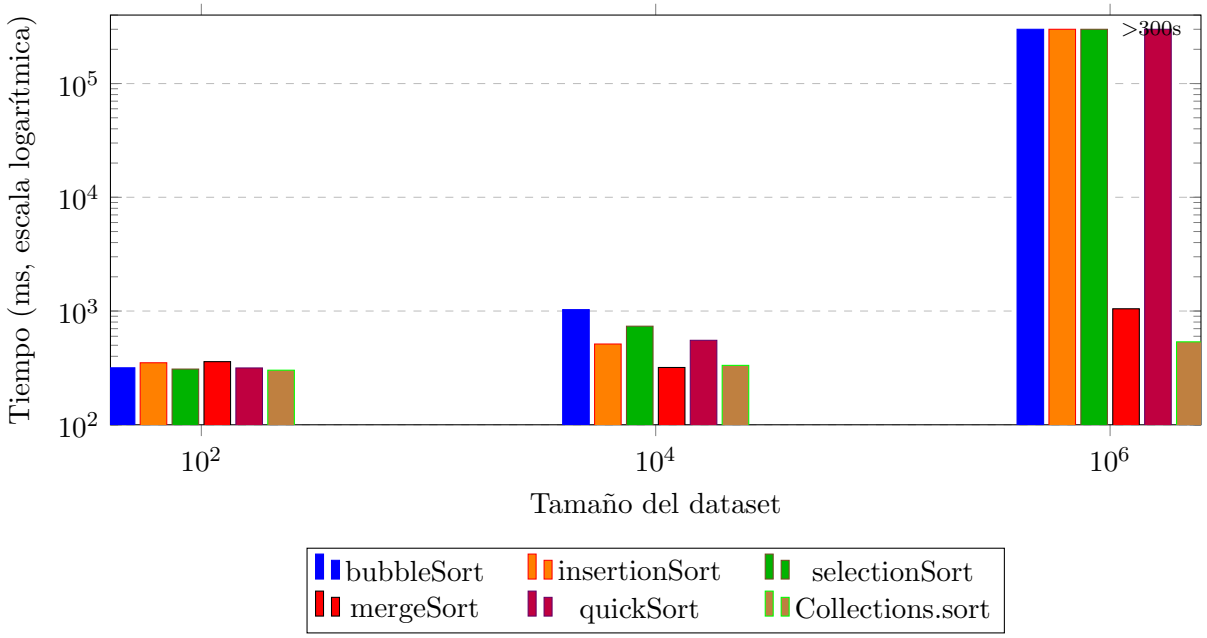


Figura 3: Comparación logarítmica de tiempos de ejecución para diferentes algoritmos de ordenamiento y tamaños de dataset al ordenar según el atributo *category*

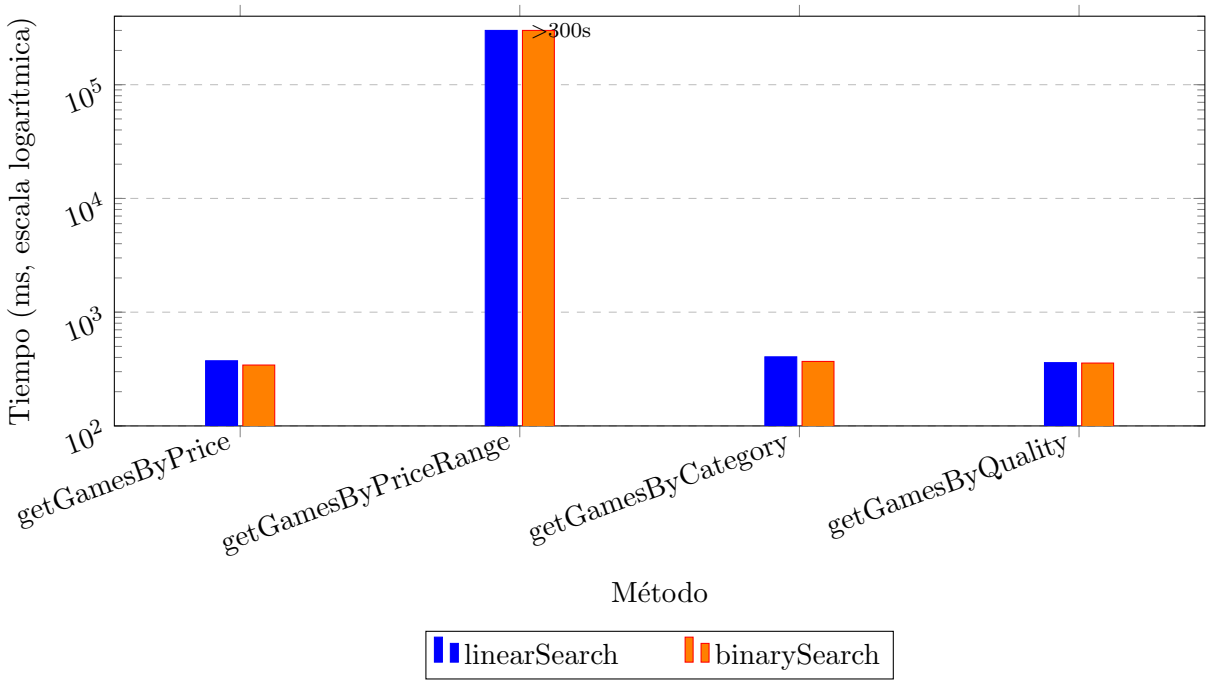


Figura 4: Comparación logarítmica de tiempos de ejecución para diferentes métodos y algoritmos de búsqueda

# 4. Análisis

## 4.1. Comparación entre búsqueda lineal y binaria

La búsqueda lineal funciona recorriendo un arreglo y verificando elemento por elemento si este cumple con la condición buscada, por tanto su complejidad siempre es de O(n), siendo n el largo del arreglo. En cambio, la búsqueda lineal funciona dividiendo el arreglo en dos arreglos según la condición, es decir, si el objetivo es menor al valor que se encuentra en la mitad del arreglo, se descarta toda la otra mitad del arreglo. Al repetir este proceso iterativa o recursivamente, el objeto que cumple con la condición buscada es encontrado con complejidad O(log n), pues va dividiendo el arreglo en subarreglos cuyo tamaño es la mitad del anterior. Sin embargo, para aplicar búsqueda binaria el arreglo debe estar ordenado, de lo contrario, los objetos que estén en una mitad no serán necesariamente menores o mayores a los de la otra mitad.

En base a los resultados obtenidos experimentalmente, se puede apreciar que el tiempo de ejecución de la búsqueda binaria siempre es levemente menor al de la búsqueda lineal, por tanto, en casos donde el arreglo a analizar está ordenado, es mejor utilizar búsqueda binaria. Sin embargo, considerando que la diferencia entre el tiempo de ejecución de la búsqueda lineal y la búsqueda binaria es despreciable en los casos estudiados, en caso de tener un arreglo sin ordenar es mejor utilizar

búsqueda lineal, pues utilizar un algoritmo de ordenamiento para posteriormente utilizar búsqueda lineal no es eficiente en términos de tiempo de ejecución.

## 4.2. Implementación de Counting Sort

Con objetivos de investigación, se implementó el algoritmo *countingSort* dentro de la clase *SearchAndSortMethods*. También, se registraron mediciones para poder comprender experimentalmente la eficiencia de dicho método. Los resultados de las mediciones son los siguientes:

Cuadro 5: Tiempos de ejecución del algoritmo countingSort para el atributo *quality*

| **Tamaño del dataset** | **Tiempo (milisegundos)** |
|---|---|
| $10^2$ | 0ms |
| $10^4$ | 2ms |
| $10^6$ | 119ms |

El tiempo de ejecución del método countingSort es considerablemente menor en comparación al resto de los algoritmos.

Cuadro 6: Comparación de tiempos de ejecución por algoritmo para el atributo *quality*

| **Algoritmo** | **$10^2$** | **$10^4$** | **$10^6$** |
|---|---|---|---|
| bubbleSort | 299ms | 716ms | Más de 300 segundos |
| insertionSort | 357ms | 486ms | Más de 300 segundos |
| selectionSort | 298ms | 469ms | Más de 300 segundos |
| mergeSort | 378ms | 360ms | 893ms |
| quickSort | 339ms | 344ms | 93524ms |
| Collections.sort | 336ms | 311ms | 469ms |
| countingSort | 0ms | 2ms | 119ms |

En base a los tiempos de ejecución registrados en el resto de algoritmos, se observa un incremento importante en la eficiencia, pues el método countingSort presenta tiempos de ejecución muy inferiores. Esta gran diferencia de tiempo de ejecución se debe a que, a diferencia del resto de algoritmos, el countingSort es un algoritmo no comparativo, por lo que en casos donde el rango a ordenar no es tan grande resulta muy eficiente. En este caso, se utilizó para ordenar juegos por su atributo *quality*, el cual es un entero que toma valores desde 0 a 100. El funcionamiento del método countingSort consiste en contar cuántas veces aparece cada valor en el arreglo a ordenar y posteriormente reconstruir el arreglo ordenado. En base a esto último, el algoritmo countingSort es especialmente útil en casos donde se necesitan ordenar datos enteros positivos cuyo valor se encuentre dentro de un rango pequeño.

Para la implementación del algoritmo countingSort, se guarda en variables el valor máximo y mínimo que toma el atributo por el cual se realizará el ordenamiento y se utilizarán para crear una variable llamada *range*, cuya función, tal como dice su nombre, es asignar el rango de la variable a ordenar, para de esta manera crear un arreglo denominado *count* que contiene la cantidad de veces que se repite cada elemento en el arreglo inicial. Posteriormente, mediante un ciclo *for*, se recorre el arreglo *count* y se acumula la cantidad de valores que son menores o iguales respecto al valor de la posición *i*. Luego, se crea un arreglo de objetos *Game* cuya función es almacenar los elementos ordenados, para ello, se utiliza un ciclo *for* en el cual se recorrerá el arreglo inicial de manera inversa y se asignarán los objetos *Game* según la calidad del *i*-ésimo objeto del arreglo inicial y el arreglo *count*. Cada vez que se asigna un objeto, se decrementa el valor del elemento posicionado en *q* - *valorMínimo*. Finalmente, se copia el arreglo de objetos en el arreglo original, para lograr un ordenamiento *in-place*.

### 4.3. Uso de Generics en Java para estructuras reutilizables

Para que la clase Dataset funcione utilizando objetos genéricos, se tendría que cambiar todas las referencias hacia game, dentro de la clase, por una variable, en este caso T. También se tendría que declarar la clase como *"public class Dataset<T>"* en vez de solo tener *"public class Dataset"*. Esto es una característica del lenguaje Java, que permite la creación de métodos, clases, etc, que trabajen con diferentes tipos de datos sin perder información valiosa o seguridad. Ventajas del uso de genéricos serían una verificación de tipos más rigurosa, ya que ahorra tiempo al evitar los errores tipográficos. Otra ventaja es la eliminación de conversiones, lo que significa que se puede usar menos código.

## 5. Conclusión

En base a los objetivos planteados y tomando en cuenta los requerimientos del programa, se puede considerar que el laboratorio fue un éxito, dado que se lograron implementar correctamente todas las clases y métodos para que el programa cumpla su función de manera consistente.

También, gracias a la realización del proyecto, se logró adquirir conocimientos prácticos acerca de cómo utilizar e implementar diversos algoritmos de búsqueda y ordenamiento. En base a las pruebas realizadas, se pueden sacar las siguientes conclusiones:

- Los algoritmos *bubbleSort*, *insertionSort* y *selectionSort* son útiles cuando la cantidad de datos a ordenar es pequeña, pues funcionan comparando de a dos elementos, lo que los hace muy ineficientes a la hora de aplicarlos a grandes cantidades de datos. Su gran ventaja es la simplicidad de sus implementaciones, pues funcionan mediante ciclos *for* o *while*, convirtiéndolos en algoritmos muy fáciles de entender.

- Los algoritmos *mergeSort* y *quickSort* son útiles cuando la cantidad de datos a ordenar es grande, pues su funcionamiento, pese a ser similar al de los algoritmos de comparación vistos previamente, sigue la estrategía algorítmica llamada *divide y vencerás*, la cual consiste en dividir recursivamente el arreglo en subarreglos más pequeños y ordenar cada uno individualmente, para finalmente unir todos los subarreglos y terminar con el arreglo principal ordenado.
- El algoritmo *countingSort* es útil cuando el dato a ordenar es un entero positivo cuyo rango es pequeño, pues funciona registrando la cantidad de veces que los elementos se repiten en el arreglo inicial.
- La búsqueda binaria, pese a tener menor complejidad teórica que la búsqueda lineal, no siempre es más eficiente que esta última, dado que para que se pueda ordenar un arreglo mediante búsqueda binaria, el arreglo debe estar ordenado. En caso de tener un arreglo sin ordenar, es más eficiente buscar un elemento mediante búsqueda lineal en vez de ordenar el arreglo y posteriormente aplicar búsqueda binaria. Sin embargo, en casos donde el arreglo a analizar ya se encuentra ordenado, es preferible utilizar búsqueda binaria.