Proyecto 1: E.D.A II. Investigación teórica: Ordenamiento Externo. Equipo 7

Andrés Basile Álvarez.
María Fernanda Martínez Jiménez.
Rodolfo Andrés Keller Ascencio.

6 de octubre del 2019

En este proyecto se realizará un programa en el que se implementen tres métodos de ordenamiento externo los cuales son: método por polifase, por mezcla equilibrada y por distribución ó Radix; para organizar el contenido de archivos de texto con extensión ".txt". Para lo anterior se requerimos definir algunos antecedentes al tema como lo son, definir el funcionamiento de los algoritmos de ordenamiento, y uso, definir los métodos de un ordenamiento externo, como funcionan y algunos ejemplos o implementaciones. Cómo manejar archivos en java (lenguaje de programación utilizado en este proyecto) y el uso de excepciones para archivos; pues así evitamos que estos no sean encontrados o leídos por el programa y esto ocasione problemas dentro del mismo. Entre algunos otros conceptos que se requieran a lo largo del proyecto.

1.Algoritmos de ordenamiento.

Un ordenamiento es un conjunto de elementos dispuestos de una determinada manera para lograr la consecución de un fin determinado.
Por lo tanto un algoritmo de ordenamiento consiste en disponer o clasificar un conjunto de datos, ya sea de una lista o de un archivo, (para efectos de este proyecto será para este último), en algún determinado orden con respecto a alguno de sus campos: orden y clave.
Orden: Relación de una elemento con respecto a otro.
Clave: Campo por el cual se ordena.
En otras palabras un algoritmo de ordenamiento provee de una nueva secuencia u acomodo a una serie de datos bajo un criterio establecido, este criterio se puede basar en su orden siendo éste de manera **ascendente** ó **descendente**.
Una lista de datos está ordenada por la clave k, si la lista está en orden con respecto a la clave anterior, este orden puede ser:

Ascendente: (i<j) entonces (k[i]<=k[j]).
Descendente: (i>j) entonces (k[i]>=k[j])

Los algoritmos de ordenamiento se clasifican según el lugar donde se realice la ordenación:

- Algoritmos de ordenamiento interno: en la memoria del ordenador.
- Algoritmos de ordenamiento externo: en un lugar externo, como un disco duro.

Y por el tiempo que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:

- Algoritmos de ordenación natural: tarda lo mínimo posible cuando la entrada está ordenada.
- Algoritmos de ordenación no natural: tarda lo mínimo posible cuando la entrada está inversamente ordenada.
- Por estabilidad: un ordenamiento estable mantiene el orden relativo que tenían originalmente los elementos con claves iguales.

Algoritmos de ordenamiento externo.

Los algoritmos de ordenamiento externo se utilizan para ordenar grandes cantidades de datos, debido a que esta información no cabe memoria principal, es necesario ocupar memoria secundaria; es decir del disco duro. El ordenamiento ocurre transfiriendo bloques más compactos de información a memoria principal en donde se ordena con algun metodo de ordenamiento interno, el cual será **mergeSort** y **quickSort** para nuestro proyecto. Posteriormente este es regresado, ya ordenado, a memoria secundaria, lo cual implica el manejo de archivos.

Estos algoritmos se usan cuando se tienen registros y/o archivos, ya que su uso es para el manejo de datos, ya sean del mismo o distinto tipo y estos pueden quedar almacenados ya ordenados.

Algunos de los algoritmos de ordenamiento externo son:

- Intercalación Simple.
- Ordenamiento Merge.
- Método de Hash.
- Método directo.
- Método natural.
- **Método por polifase.**
- **Método por Mezcla equilibrada.**
- **Método por distribución (Radix).**

Método por polifase.

El método por polifase usa "m" archivos auxiliares para ordenar "n" registros de un archivo, este considera un archivo de salida y otros "m-1" archivos de entrada.

Archivos de entrada: son aquellos que contendrán la información para realizar el ordenamiento. Archivos de salida: son aquellos donde se almacenan los registros ordenados.

Durante el proceso, cuando se realiza el registro del archivo de entrada este se convierte en uno de salida, el anterior de salida pasa a ser de entrada y el ordenamiento continua, el número de iteraciones continua hasta que solo se tenga un bloque ordenado y se devuelva al archivo.

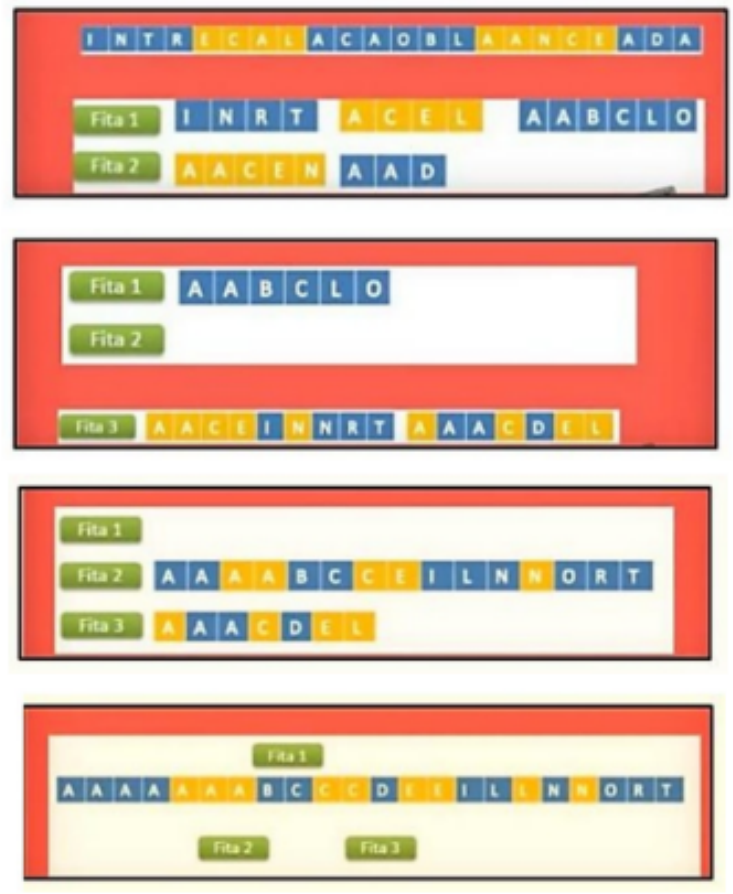
Mientras existan datos de entrada o los bloques no esten vacios, los pasos a seguir son:

1. Leer 'm' elementos.
2. Ordenar los elementos por método un interno (en este caso quickSort).
3. Si los 'm' elementos anteriores se colocaron en un archivo 1, los siguientes 'm' elementos se colocan en un segundo archivo.

4. Se intercala el primer bloque del primer archivo con el primer bloque del segundo y este nuevo bloque se devuelve a un archivo número 3.

5. Se repite este paso hasta que los archivos auxiliares ya no tengan más elementos por intercalar.

Ejemplo esquematizado:



Método por Mezcla equilibrada.

Usa "n" archivos auxiliares de los cuales la mitad son de entrada y la mitad de salida. Inicia distribuyendo los bloques del archivo origen en la primera porción de archivos auxiliares. A partir de esta, repite los procesos de mezcla reduciendo a la mitad el número de bloques hasta que queda un único bloque. Así, el proceso de mezcla se realiza en una sola fase en lugar de dos fases; separación y fusión.

Su algoritmo funciona de la siguiente forma:

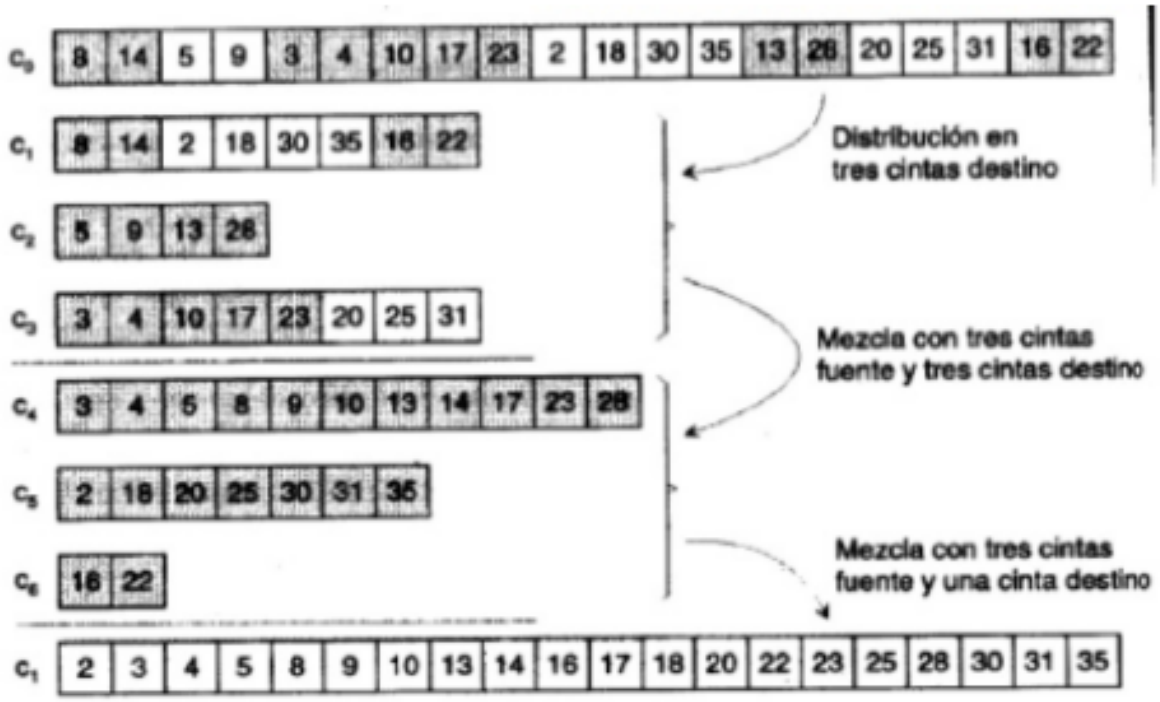
1. Distribuye registros del archivo original por bloques en la primera mitad de archivos auxiliares. A continuación estos se consideran archivos de entrada.

2. Mezcla bloques de la mitad de archivos de entrada y los escribe consecutivamente en la otra mitad que corresponde a los archivos de salida, es decir los ordena, en este caso mediante mergeSort.

3. Cambia la finalidad de los archivos, los de entrada pasan a ser de salida y viceversa.

4. Repite a partir del segundo paso hasta que queda un único bloque, solo ahí la colección está ordenada completamente.

Ejemplo esquematizado:



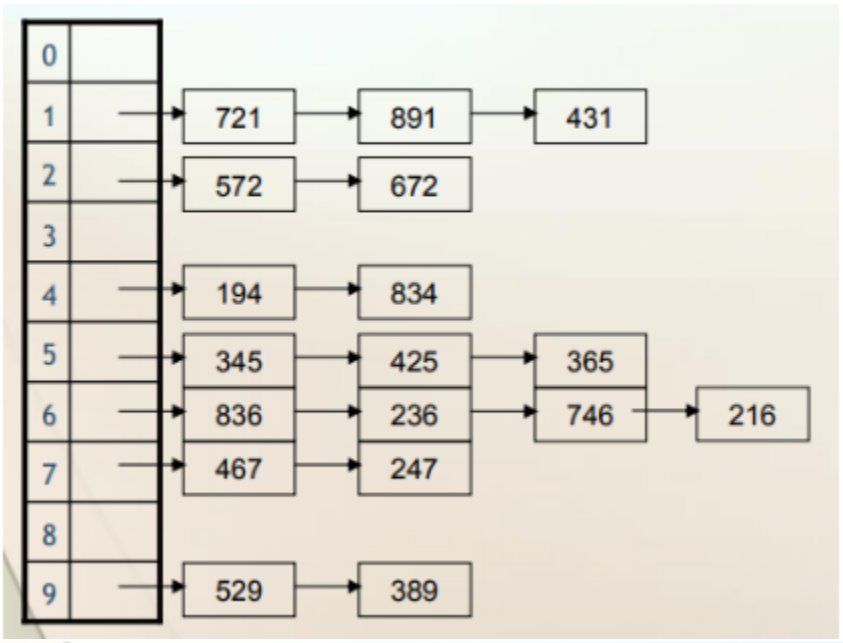
Método por distribución (Radix).

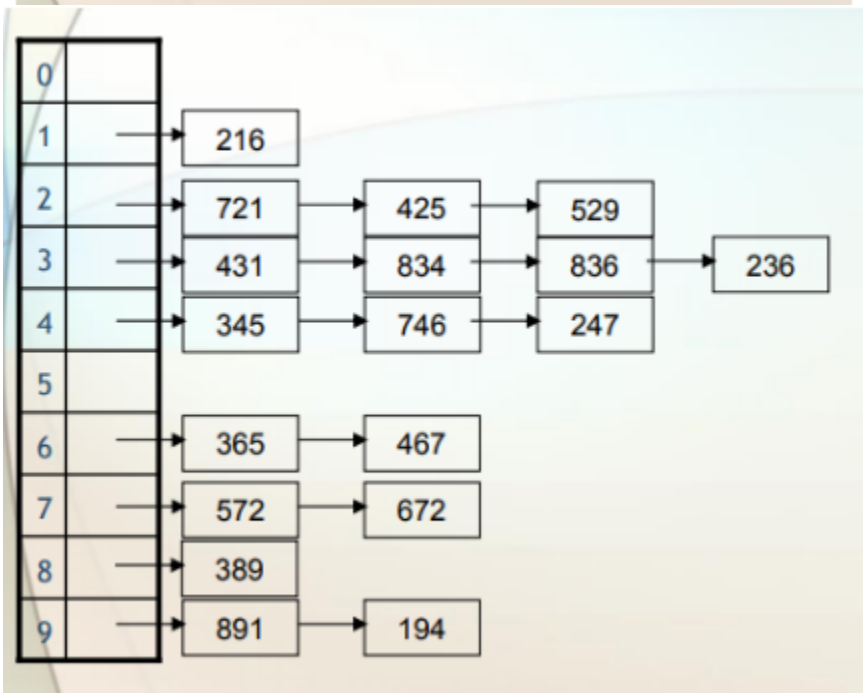
Este método se basa en los valores absolutos de los dígitos de los números a ordenar, consiste en lo siguiente:

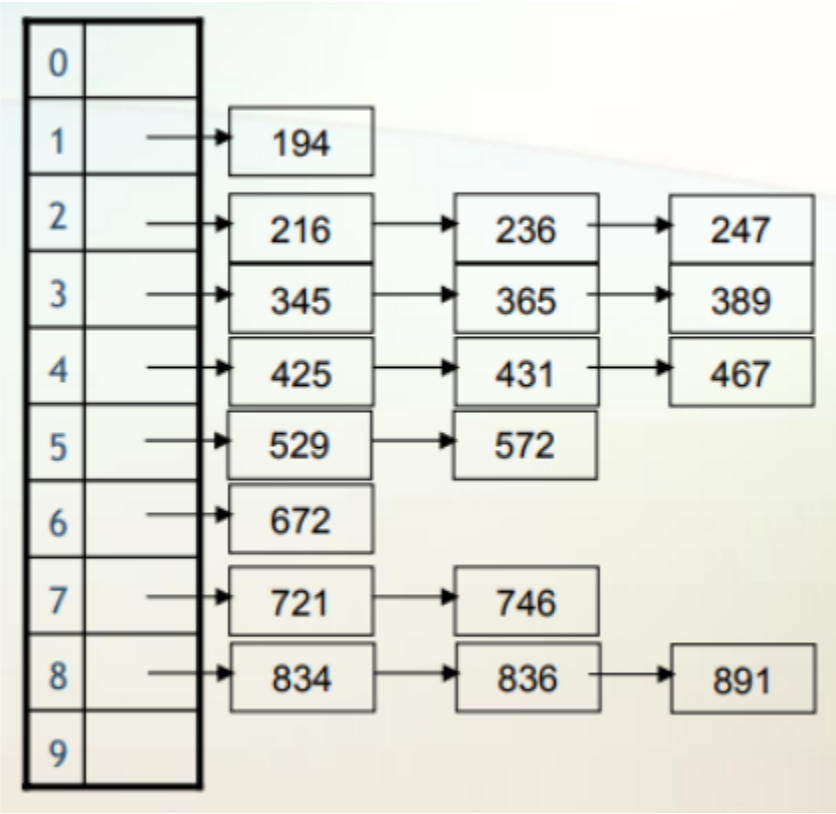
- Crea diversas colas de elementos, cada una caracterizada por tener en sus componentes un mismo dígito (letra si es alfabética) en la misma posición; estas colas se ordenan en orden ascendente y se reparten de nuevo en colas según el siguiente dígito de la clave.

- La idea clave es clasificar por colas primero respecto al dígito de menor peso (menos significativo) d1, después concatenar las colas, clasificar de nuevo respecto al siguiente dígito d2, y así sucesivamente se sigue con el siguiente dígito hasta alcanzar el dígito más significativo dn. En ese momento la secuencia estará ordenada.

Ejemplo:
Se requiere oirdenar los siguientes elementos:
345,721,425,572,836,467,672,194,365,236,891,746,431,834,247,529,216,389.


0
1
721
891
431
2
572
672
3
4
194
834
5
345
425
365
6
836
236
746
216
7
467
247
8
9
529
389


0
1
216
2
721
425
529
3
431
834
836
236
4
345
746
247
5
6
365
467
7
572
672
8
389
9
891
194



Finalmente el conjunto de elementos que se devolverá al archivo es:
194, 216, 236, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 721, 746, 834, 836, 891.

Algoritmos de ordenamiento interno.

Debido a que los bloques de información extraídos de los archivos entran a la memoria principal para ordenarlos, aquí se utilizan algoritmos de ordenamiento interno. Para lo anterior existen varios métodos para ordenar los elementos del bloque insertado:

- Insertion sort.
- Selection sort.
- Bubble sort.
- Shell sort.
- Merge sort.
- Quick-sort.
- Heap sort.
- Radix sort.
- Address-calculation

Entre algunos otros, pero para nuestro contexto y proyecto utilizaremos, **Merge sort** y **Quick sort**.

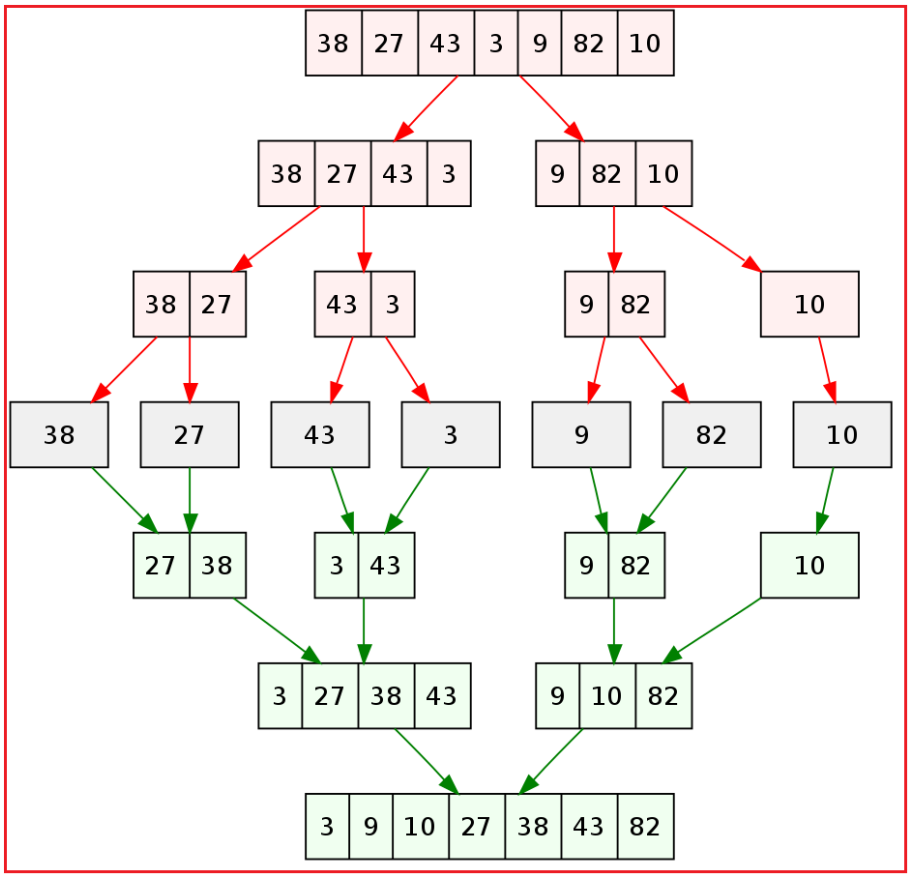
Merge sort.

En éste método se unen dos estructuras ordenadas para formar una sola ordenada correctamente.
Tiene la ventaja de que utiliza un tiempo proporcional a n log (n), su desventaja radica en que se requiere de un espacio extra para el procedimiento, pero debido a que se utiliza espacio extra ya en el almacenamiento externo de los archivos este requerimiento de memoria no simboliza un gran problema.
Este tipo de ordenamiento es útil cuando se tiene una estructura ordenada y los nuevos datos a añadir se almacenan en una estructura temporal para después agregarlos a la estructura original, en este caso nuestro archivo original, de manera que vuelva a quedar ordenada; por ello este ordenamiento fue elegido para uno de nuestros ordenamientos, Mezcla equilibrada.

Funcionamiento:

Si tenemos dos archivos que ya están ordenados, podemos mezclarlos para formar un solo archivo ordenado en tiempo proporcional a la suma de los tamaños de los dos archivos.
Esto se hace leyendo el primer elemento de cada archivo, copiando hacia la salida al menor de los dos, y avanzando al siguiente elemento en el archivo respectivo. Cuando uno de los dos archivos se vacía, todos los elementos restantes del otro se copian hacia la salida. Como cada elemento se copia sólo una vez, y con cada comparación se copia algún elemento, el costo de mezclar los dos archivos es lineal. Usando esta idea en forma reiterada se ordena el conjunto.

Ejemplo esquematizado:

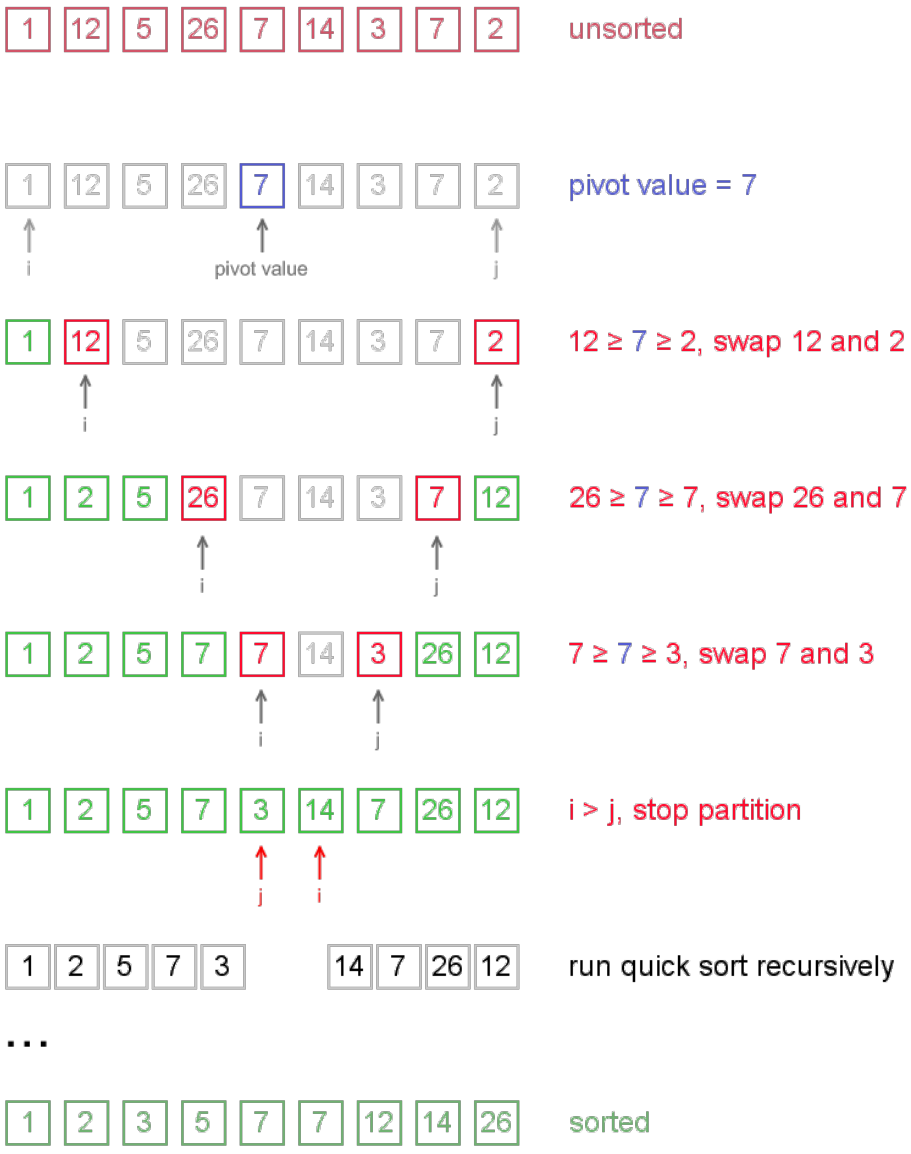


Quick sort.

Este ordenamiento, es un algoritmo basado en la técnica de divide y vencerás, que permite, ordenar n elementos en un tiempo proporcional a n log (n).
El algoritmo se basa fundamentalmente en lo siguiente:

1. Se elige un elemento de la lista de elementos a ordenar, que se llamará pivote.
2. Se reubican los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. De manera que la lista quedará separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
3. Se repite este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

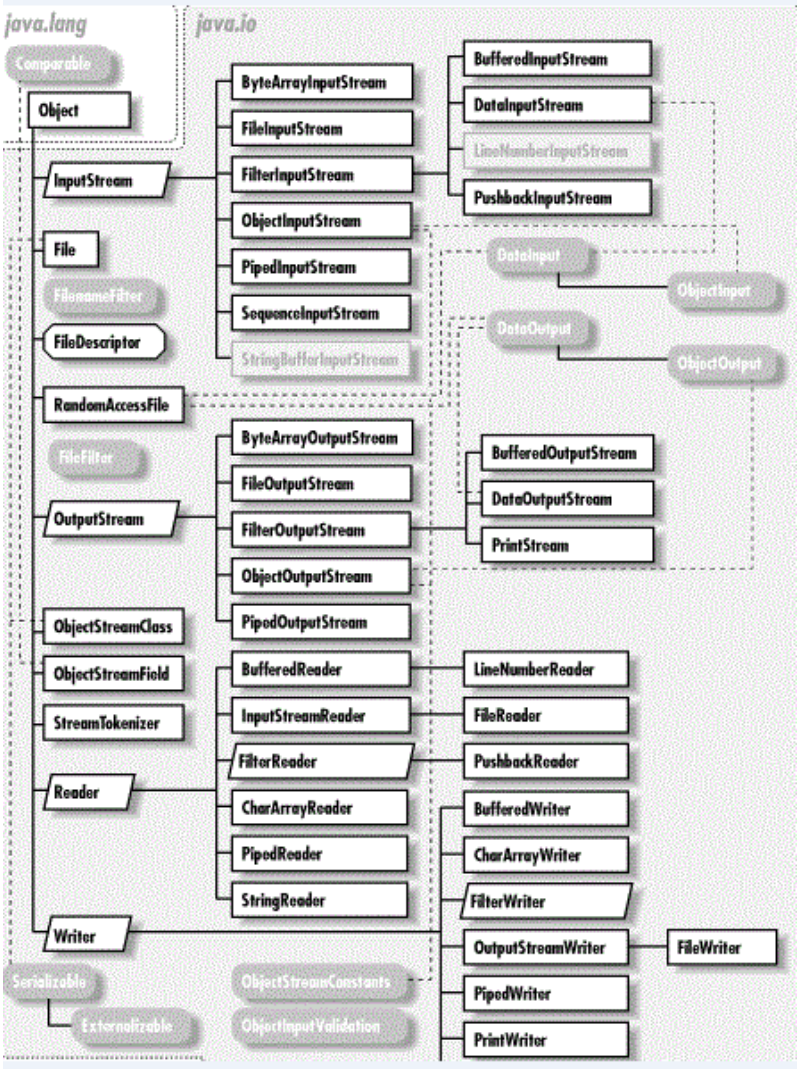
Ejemplo esquematizado:



2. Manejo de archivos en java.

Los datos que se crean en un programa solamente existen durante la ejecución de los mismos, pues las variables que los contienen liberan la memoria de manera automática cuando el programa termina.
Es por ello que el principal objetivo del uso de archivos dentro de un programa es que al salir de este, los datos generados queden almacenados en algún lugar que permita su recuperación desde él mismo u otros programas. Esto consiste en organizar esa información en uno o varios archivos almacenados en algún soporte de almacenamiento persistente, como lo son: los discos duros, discos magnéticos, discos ópticos, cintas magnéticas, tarjetas de memoria, etc. Otra posibilidad es el uso de bases de datos que utilizan archivos como soporte para el almacenamiento de la información.

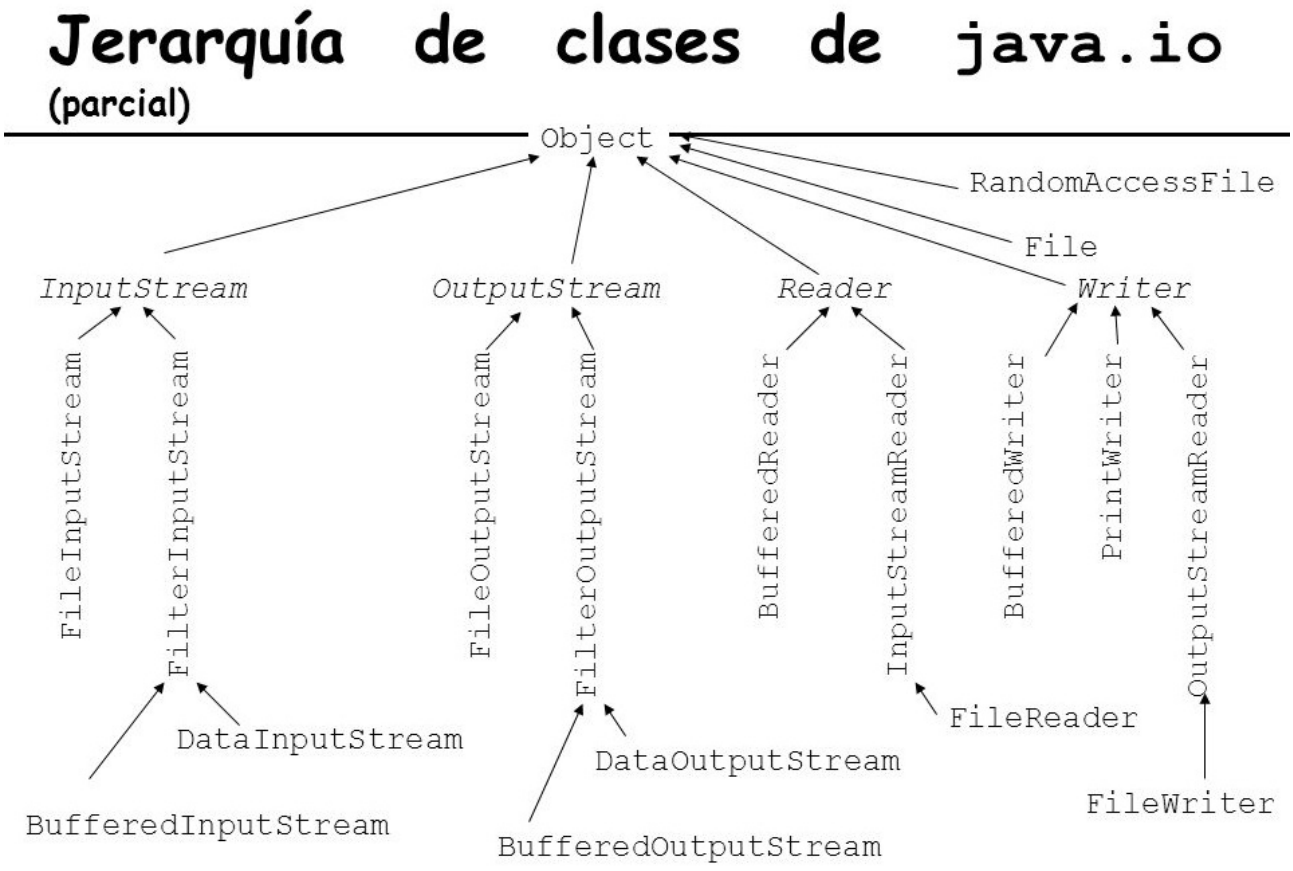
En Java, los distintos tipos de archivos se diferencían por las clases que usan para representarlos y manipularlos. Como las clases que se usan pertenecen a la biblioteca estándar del lenguaje, su uso es algo más complejo que, ya que su diseño se ha realizado pensando en un uso industrial. Las clases usadas para el tratamiento de archivos están ubicadas en el paquete java.io por lo que estas, deben ser importadas. En el siguiente esquema se muestran las clases provenientes de este paquete:



Cuando el código trabaja con archivos, se tiene que considerar que pueden existir diversos problemas cuando se trabaja con ellos, por ejemplo: puede desconectarse el pendrive a medio ejecutar del programa, es un disco en red y ésta ha caído, no tiene más espacio para almacenar información, el archivo puede no existir, el archivo puede cerrarse durante la lectura del mismo, etc.
Es por ello que, se debe introducir un mecanismo estándar en Java para tratar con estos y otros errores que pueden darse en un programa, este conjunto de posibles errores se les conoce como las excepciones, que veremos más adelante en esta investigación.

Existen diversas clases dedicadas a leer datos de una fuente de entrada como lo es un archivo, conocidas como clases para flujos de entrada. Para leer un flujo de caracteres se utilizan las clases: **BufferReader**, **FileReader**.
Así como se requiere leer datos también se requiere enviar flujos de datos a dispositivos de salida; es decir escribir los datos en un archivo, para ello se utilizan las clases para flujo de salida: PrintWriter y **FileWriter**.

Ambas clases, tanto las de lectura como las de envio contenidas en el paquete **java.io**, son clases derivadas de las clases Writer y Reader que a su vez derivan de la clase Object. Esta jerarquia se esquematizan de la siguiente forma:



Para nuestra implementación fueron usadas: **FileReader**,**BufferedReader** y **FileWriter**.

FileReader.

Permite que leer archivos almacenados usando los objetos File o Blob dependiendo de los datos que se pretenden leer.
Se define de la siguiente forma:

```
FileReader fr = new FileReader(file)
```

Es una clase que opera a bajo nivel. El método read lee todo el archivo, devuelve el tamaño y lo devuelve en un array.
Sus métodos están definidos en la clase InputStreamReader , algunos de los más importantes son:

- **abort()**: Interrumpe la operación de lectura.
- **readAsArrayBuffer()**:Lee el contenido del objeto especificado, una vez terminado, el atributo result contiene un ArrayBuffer representando los datos del archivo.
- **readAsBinaryString()**: Lee el contenido del objeto especificado, una vez terminado, el atributo result contiene los datos binarios del archivo como una cadena.
- **readAsDataURL()**: Lee el contenido del objeto especificado, una vez terminado, el atributo result contiene un data: URL que representa los datos del fichero.
- **readAsText()**: Lee el contenido del objeto especificado, una vez terminado, el atributo result contiene el contenido del archivo como una cadena de texto.

BufferedReader.

BufferedReader es una clase de Java para leer el texto de una secuencia de entrada (como un archivo) almacenado en el búfer (espacio de la memoria en un disco duro o en un instrumento digital, que se utiliza para almacenar datos temporalmente), utiliza un búfer para mejorar eficiencia de lectura de caracteres, este lee caracteres, matrices o líneas.

Se define de la siguiente forma:

```
InputStreamReader entrada = new InputStreamReader(System.in);
BufferedReader teclado = new BufferedReader (entrada);
String cadena = teclado.readLine();
```

En la siguiente tabla se encuentran algunos de los métodos que emplea esta clase:

Método	Tipo de Retorno	Descripción
`mark(int limiteCaracteres)`	`void`	Marca la posición actual en la que se encuentre el apuntador en el flujo, el parámetro del método indica el número de caracteres que pueden ser leídos mientras se mantenga esta marca.
`read()`	`void`	Lee un solo carácter del flujo, lo curioso de este método es que retorna un número entero, ¿Por qué? Más adelante la respuesta.
`readLine()`	`String`	Lee una línea completa de texto.
`ready()`	`boolean`	Este método es utilizado para saber si aún hay caracteres en el flujo para ser leídos, detalles más adelante.
`reset()`	`void`	Reinicia el flujo hasta la marca más reciente que se haya hecho.
`skip(long n)`	`long`	Mueve el apuntador del flujo las posiciones necesarios para evitar la cantidad de caracteres de n.

FileWriter.

La clase FileWriter permite escribir en un archivo. Para crear objetos FileWriter se pueden utilizar los constructores:

```
FileWriter(String path)
FileWriter(File objetoFile);
```

El archivo se crea y si ya existe su contenido se pierde. Sin embargo para abrir un archivo de texto existente sin perder su contenido y añadir más contenido al final se utilizan los constructores:

```
FileWriter(String path, boolean append)
FileWriter(File objetoFile, boolean append)
```

Si el parámetro append es true significa que los datos se van a añadir a los existentes. Si es false los datos existentes se pierden.
La clase FileWriter proporciona el método write() para escribir cadenas de caracteres aunque lo normal es utilizar esta clase junto con la clase PrintWriter para facilitar la escritura.

El objeto para escribir en el archivo se define de la siguiente forma:

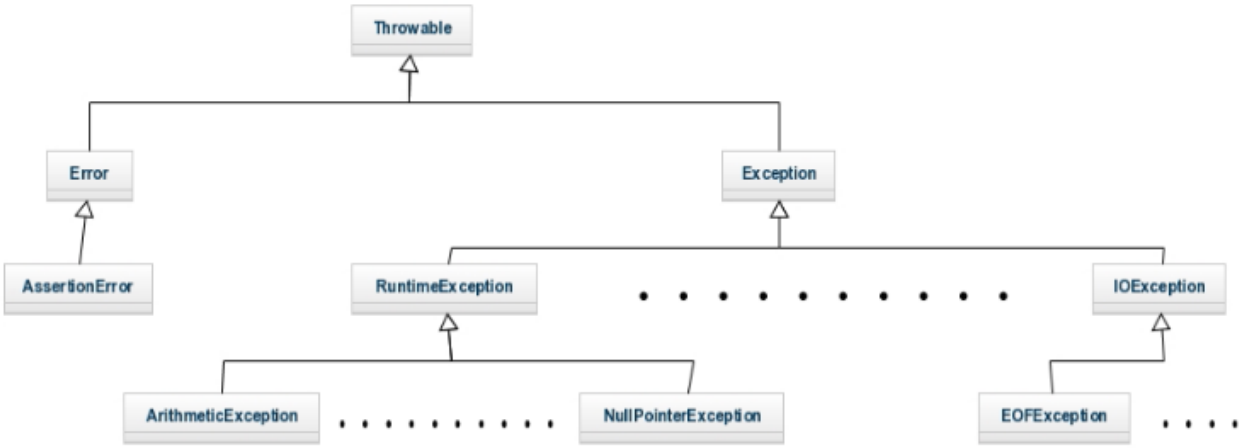
```
FileWriter fw = new FileWriter("c:/ficheros/datos.txt");
```

3. Manejo de excepciones.

Las excepciones son un mecanismo que permite a los métodos indicar que algo "anómalo" ha sucedido y que impide su correcto funcionamiento, de manera que programador pueda detectar la situación errónea. Cuando esto ocurre dice entonces, que el método ha lanzado **throw**; una excepción.
Cuando esto sucede, en vez de seguir con la ejecución normal de instrucciones, se busca hacia atrás en la secuencia de llamadas si hay alguna que quiera atraparla, hacer **catch**. Si ninguna de las llamadas decide atraparla, el programa acaba su ejecución y se informa al usuario del error que se ha producido la excepción y que nadie ha podido tratarlo.
Muchas de las excepciones que existen en Java, son excepciones en tiempo de ejecución y no obligan a que el programador las trate explícitamente. Sin embargo, en Java, existe otro tipo de excepciones, las denominadas excepciones comprobadas, que obligan al programador que dentro del código de un método invoque una instrucción que puede lanzarla a:

- Atrapar dicha excepción, colocando dicha instrucción en un bloque **try-catch**; ó:
- Declarar en la cabecera del método, que dicho método puede lanzar esa excepción, usando una declaración **throws**.

Cuando en Java se produce una excepción, se crea un objeto de una determinada clase (dependiendo del tipo de error que se haya producido), que mantendrá la información sobre el error producido y proporcionará los métodos necesarios para obtener dicha información. Estas clases tienen como clase padre la clase **Throwable**.
En el siguiente esquema se muestran algunas de estas clases, pero existen muchísimas más excepciones que las que se muestran:



El tratamiento simplificado las excepciones mismas, consiste en lo siguiente.

El código que manipula los archivos, este tiene la siguiente estructura:

```
1 try {
2   Codigo que abre y trata el fichero
3 } catch (IOExcepction ex) {
4   Código que trata el error
5 }
```

La idea de esta construcción es intentar (try) ejecutar esas instrucciones y, en caso de producirse un error en el tratamiento de los archivos es decir, una vez se haya lanzado una IOException, atrape (catch) ese error y ejecute el código de corrección. En el código de corrección se puede solamente escribir el mensaje de error.
Si en vez de tratar el error se quiere indicar que el método puede lanzar excepciones, en su cabecera se debe poner:

```
1 public int methodThatCanThrow(params) throws IOException {
2
3   Código que trata ficheros pero no atrapa IOException
4
5 }
```

Bibliografía:

- Algoritmos y Estructuras de Datos: Ordenación. Recuperado el 26 septiembre, 2019, de https://users.dcc.uchile.cl/
- Definicion, Definicion de Ordenamiento. Recuperado el 26 septiembre, 2019, de https://definicion.mx/ordenamiento/
- TPM — Tutorial de Programacion Multiplataforma. (2014, 27 mayo). Recuperado el 26 septiembre, 2019, de https://itslr.edu.mx/archivos2013/TPM/temas/s3u5.html
- D.P, Arturo. Análisis y Complejidad de Algoritmos, Métodos de ordenamiento.Recuperado el 26 septiembre, 2019, de https://delta.cs.cinvestav.mx/ adiaz/anadis/Sorting2.pdf
- Métodos de Ordenamiento, Unidad VI: Estructura de datos. Recuperado el 26 septiembre, 2019, de http://mapaches.itz.edu.mx/ mbarajas/edinf/Ordenamiento.pdf
- Merge Sort Algorithm in C. (2019, 5 octubre). Recuperado el 26 septiembre, 2019, de https://dotnettutorials.net/lesson/merge-sort-algorithm-in-csharp/
- Abad, F., Fajardo, P., Muñoz, C. Universidad de Cuenca, Facultad de ingeniería. (2017, 12 junio). Algoritmos de ordenamiento externo. Recuperado el 26 septiembre de 2019, de https://dspace.ucuenca.edu.ec
- Ordenamiento Rápido (Quicksort). Recuperado el 26 septiembre, 2019, de http://mis-algoritmos.com/ordenamiento-rapido-quicksort
- QUICKSORT (Java, C++) — Algorithms and Data Structures. Recuperado el 26 septiembre, 2019, de http://www.algolist.net/Algorithms/Sorting/Quicksort
- J.M.Gimeno y J.L.González, Manejo de Archivos en Java. Recuperado el 28 septiembre, 2019, de http://ocw.udl.cat/enginyeria-i-arquitectura/programacio-2/continguts-1/4-manejo-bai81sico-de-archivos-en-java.pdf
- Cómo leer un archivo en Java: Ejemplo de BufferedReader. (2018, 19 marzo). Recuperado el 28 septiembre, 2019, de https://guru99.es/buffered-reader-in-java/
- El lenguaje de programación Java - Video online. (2017, 5 diciembre). Recuperado el 28 septiembre, 2019, de , 2019, de https://slideplayer.es/slide/3789371/
- Lic. T. V. Nancy, M. Manejo de archivos Recuperado el 28 septiembre, 2019, de , 2019, de https://es.slideshare.net/chicaGeekGPLMX/manejo-de-archivos-en-java
- FileReader. (2019, 24 marzo). Recuperado el 4 octubre, 2019, de https://developer.mozilla.org/es/docs/Web/API/FileReader
- Ficheros de texto en Java. Recuperado el 4 octubre, 2019, de http://puntocomnoesunlenguaje.blogspot.com/2013/05/ficheros-de-texto-en-java.html
- Ricardo., M. (2017, 2 junio). Excepciones (Exception) en Java, con ejemplos - Jarroba. Recuperado el 4 octubre, 2019, de https://jarroba.com/excepciones-exception-en-java-con-ejemplos/

Análisis.

Análisis de los algoritmos de ordenamiento por mezcla equilibrada, polifase y distribución.

Mezcla equilibrada.

La mezcla equilibrada es un tipo de algoritmo de ordenamiento externo (utiliza memoria secundaria) en el cual se realizan particiones de los elementos del archivo original tomando secuencias de máxima longitud. Este algoritmo de ordenamiento surge como una optimización de la mezcla directa (mezcla de particiones de tamaño variable [1,2,4,8, ...]).

La implementación del algoritmo de mezcla equilibrada en un lenguaje de programación (en este caso Java) requiere del manejo de archivos, para lo cual utilizamos las clases **FileReader**, **BufferedReader** y **FileWriter**.

Al comienzo de la clase **MezclaEquilibrada**, se definió el constructor para crear objetos de esta clase. Al constructor se le pasa como atributo el nombre del archivo (con extensión .txt) y un entero que permite identificar el tipo de ordenamiento que se va a realizar (donde 1 significa **ordenamiento ascendente** y 2 significa **ordenamiento descendente**).

Dentro del constructor, y como en prácticamente todos los métodos de las clases, se debió utilizar el manejo de excepciones con **try-catch**. Para el caso del constructor, se utiliza el catch para obtener todas las excepciones de tipo **IOException** (**de entrada o salida de datos**) que puedan ocurrir y enviar un mensaje de error.

El constructor sirve para leer el archivo original, el cual se quiere ordenar, y guardar los números que se encuentran dentro en el arreglo **numeros** al separar la lectura del archivo por comas. Luego de leer todos los elementos, mediante un ciclo **for**, se pasan los elementos de tipo String del arreglo **numeros** al ArrayList **numerosDouble**, ahora como tipo Double.

Pasando de la primera lectura del archivo original, se entra al método **bloques**. Este método es de tipo **void** y comienza por crear dos nuevos archivos para escritura utilizando la clase **FileWriter**. Estos dos archivos serán los archivos auxiliares que permitirán el correcto funcionamiento del ordenamiento.

En **bloques**, se utiliza un ciclo **while** que verifique que el ArrayList **numerosDouble** contenga elementos. Dentro de éste, se utiliza otro ciclo **while** con la condición de que el número en la posición "contador+1" sea mayor que el número en la posición "contador", con **contador** = 0. En caso de que se cumpla esta condición (que el número en la posición 1 sea mayor que el de la posición 0), querría decir que nos encontramos con una **natural run** o secuencia natural de números. Por lo tanto, se puede remover el elemento de la posición 0 y escribirlo sobre el primer archivo auxiliar. Una vez que termine el ciclo **while**, se escribe el último elemento (de la posición **contador**) en el primer auxiliar porque querría decir que allí termina la secuencia natural de números.

Luego, siempre y cuando el tamaño del ArrayList **numerosDouble** siga siendo mayor a uno, se realiza el mismo procedimiento de verificación de **natural runs** y ahora se escriben en el segun-

do archivo auxiliar. Esto se repite (intercalando el archivo auxiliar donde se escriben las secuencias naturales) hasta que el ArrayList **numerosDouble** ya no tenga elementos. Justo al terminar con la división en bloques de secuencias naturales, se borra el contenido del archivo original para poder escribir sobre él y continuar el procedimiento de ordenamiento.

A continuación, se presenta el funcionamiento del método bloques para separar las secuencias naturales de números que se puedan presentar en el archivo original:

```
FileWriter archivoOriginal = new FileWriter(nombreArchivoOriginal,false);
FileWriter aux1 = new FileWriter(nomAux1);
FileWriter aux2 = new FileWriter(nomAux2);

int contador = 0;
int termina = 1;
while(1<numerosDouble.size()){
        while(numerosDouble.size()>1&&numerosDouble.get(contador+1)>numerosDouble.get(contador)&&numerosDouble.get(contador+1)!=null){
            aux1.write(numerosDouble.remove(contador).toString());
            aux1.write(",");
            termina=1;
        }
        aux1.write(numerosDouble.remove(contador).toString());
        aux1.write("/");
        termina=2;
    if(1<numerosDouble.size()){
        while(numerosDouble.size()>1&&numerosDouble.get(contador+1)>numerosDouble.get(contador)){
            aux2.write(numerosDouble.remove(contador).toString());
            aux2.write(",");
            termina=2;
        }
        aux2.write(numerosDouble.remove(contador).toString());
        aux2.write("/");
        termina=1;
    }
}
if(!numerosDouble.isEmpty()){
    if(termina == 1){
        aux1.write(numerosDouble.remove(contador).toString());
        aux1.write("/");
    }
    if(termina == 2){
        aux2.write(numerosDouble.remove(contador).toString());
        aux2.write("/");
    }
}
aux1.close();
aux2.close();
archivoOriginal.write("");
archivoOriginal.close();
```

Luego de la separación en bloques del archivo original, el procedimiento de ordenamiento por mezcla equilibrada continúa mediante un ciclo **do-while** en donde se realiza el proceso que se expondrá a continuación hasta que el método **verificaOrdenamiento** devuelva un valor **true**, indicando que el archivo original se encuentra ordenado. A continuación, se presenta el proceso dentro del ciclo **do-while**.

El primer método al que se llama es **lecturaArchivoAuxiliar** (primero para el primer archivo auxiliar e inmediatamente después para el segundo archivo auxiliar). Este método con retorno de tipo ArrayList¡String¿toma como parámetro el nombre del archivo auxiliar que se va a leer así como el nombre con el cual se imprimirá más tarde el bloque al que pertenece.

Dentro de **lecturaArchivoAuxiliar**, se crea un objeto de la clase **FileReader** con su respectivo **BufferedReader** y se utiliza un ciclo **while** para separar cada uno de los bloques generados en el método anterior y poder trabajar directamente con cada uno de ellos. Luego de que se separan

los bloques del archivo auxiliar (1 o 2) y se guarda en el arreglo **bloque**, se pasa cada uno de los elementos (bloques completos de secuencias naturales de números) en el arreglo **bloqueAux**, pero ahora separando los números por comas para, mediante un ciclo for, poder almacenar los elementos separados en el ArrayList **bloqueArchivoString**, el cual contendrá todos los números de los bloques de un archivo auxiliar y colocará una diagonal "/" para mostrar la finalización de la lectura de todos los bloques. El ArrayList devuelto por el método es justamente **bloqueArchivoString**.

De esta manera, ya se logró leer correctamente todo el archivo original, dividirlo en bloques de secuencias naturales de máxima longitud de números y pasar estos bloques leídos de ambos archivos auxiliares a ArrayList para su posterior manejo.

El siguiente método que se debe realizar es**escrituraBloquesArchivoOriginal**. Este método, a grandes rasgos, permite leer los ArrayList generados en el método anterior y realizar un "merge" de ambos ArrayList (bloque por bloque) para escribirlos nuevamente en el archivo original. El método comienza declarando un objeto de tipo **fileWriter** y, mientras haya elementos en el **bloqueArchivo1String** y en **bloqueArchivo2String** (ArrayList con los elementos de los bloques pertenecientes a cada uno de los archivos auxiliares), se comprobará cuál de los elementos de los ArrayList es menor para poder realizar el "merge" o unión de ambos y colocarlo en el archivo original.

Entonces, lo que realiza este método es tomar el primer bloque del archivo auxiliar 1 y unirlo con el primer bloque del archivo auxiliar 2, colocando la unión como un único bloque en el archivo original. Continuando de esta manera hasta que ya no existan elementos en los bloques del archivo auxiliar 1 y 2, momento en el cual se escribirá una diagonal "/" que indique la finalización de la primera unión de bloques.

Terminado la unión de los bloques en el archivo original, se debe volver a leer el archivo original para pasar los elementos de nueva cuenta a los archivos auxiliares. Ello se logra mediante el método **leerArchivoOriginalPasarAAuxiliares**. Dentro de este método, se realiza una lectura del archivo original similar a la que se realizó en el método bloques. Sin embargo, se vuelve a dividir por "/" y luego por ",". De esta manera, se logra leer los **bloques** que fueron unidos (del archivo auxiliar 1 y 2) y colocados en el original para que, mediante el uso de ciclos **while**, se puedan leer estos bloques y colocarlos nuevamente en los archivos auxiliares. Cada vez que se escriba la unión de los bloques generada en el método anterior, se vuelve a escribir una diagonal "/" para identificar el final del nuevo bloque escrito en cada archivo auxiliar.

Finalilzada la nueva escritura de bloques en los archivos auxiliares, se utiliza el método **lecturaArchivoOriginal** para leer los bloques que quedaron definidos en el archivo original. Por último, se utiliza el método **verificaOrdenamiento**, el cual utiliza el bloque generado a partir de la última lectura del archivo original para comprobar la cantidad de diagonales "/" que se encuentran en el archivo original. Esto debido a que cuando la cantidad de líneas sea igual a 1, sólo existiría un bloque en el archivo original que contenga a todos los números ya ordenados y, por tanto, el **boolean** de ordenamiento sería igual a **true**.

Al repetir los métodos mencionados anteriormente, hasta que el valor de **verificaOrdenamiento** sea **true**, podemos afirmar que el archivo quedará ordenado de manera correcta (en orden ascendente). Sin embargo, si se quisiera realizar el ordenamiento de manera descendente, se debería de

realizar una partición en bloques buscando secuencias naturales descendentes en el archivo original, logrado mediante **bloquesDescendente**. Luego, se continuaría con un proceso similar al ascendente con la única diferencia de que la escritura de los bloques en el archivo original cambiaría debido a que estos se encontrarán en orden descendente.

Polifase.

Polifase es un método de ordenamiento externo que consiste en aplicar una estrategia de generación de bloques mientras se realiza la lectura del archivo original. Consiste en dos fases y utiliza 3 archivos auxiliares para su correcto funcionamiento.

La primera fase del ordenamiento consiste en la lectura del archivo original, separando en bloques de "n" cantidad de claves. Se utiliza un ordenamiento interno (en este caso **Quicksort**) para ordenar el bloque y se coloca en el primer archivo auxiliar. Luego, se repite este procedimiento, pero se coloca el bloque ordenado en el archivo auxiliar 2, intercalando los siguientes bloques entre los archivos 1 y 2.

Luego, se realiza la intercalación de los bloques (primero del archivo auxiliar 1 con el primero del archivo auxiliar 2 y se deja el resultado en el original. Luego, se intercala el siguiente bloque de cada archivo auxiliar y se deja el resultado en un tercer archivo auxiliar.

Para poder implementar el método de ordenamiento por polifase en Java, se debe, al igual que con cualquiera de los otros algoritmos de ordenamiento externo realizados, conocer el funcionamiento de las clases que hacen posible el manejo de archivos en Java.

El comienzo del ordenamiento por polifase es igual que el ordenamiento por mezcla equilibrada. El constructor de esta clase recibe como parámetro el nombre del archivo que se quiere ordenar, el tipo de ordenamiento a realizar (1 para ordenamiento ascendente y 2 para ordenamiento descendente) y el número de claves o números **"n"** en los cuales se dividirá cada uno de los bloques generados.

En el constructor, se lee el archivo original y se pasan los **números** al arreglo números de String separando la lectura por comas ",". Luego, mediante un ciclo **for**, se pasan las cadenas con los números de **numeros** a **numerosDouble**, cambiando su tipo a Double en un ArrayList. Al igual que en mezcla equilibrada y que en cualquier programa de Java que maneje archivos, es importante utilizar el manejo de excepciones con **try-catch** para manejar los errores que puedan ocurrir.

Después de leer el archivo original y guardar sus elementos en el ArrayList, inicia la primera fase de ordenamiento ascendente en **fase1BloquesAscendente**. En ella, se crean dos objetos de tipo **FileWriter**, uno para cada uno de los archivos auxiliares 1 y 2. Dentro de un ciclo **while**, se recorre el ArrayList **numerosDouble** para dividirlo en bloques según el número de claves por bloque definido por el usuario. Se agregan las claves a un bloque **numerosDouble1** y se utiliza un algoritmo de ordenamiento interno, en este caso **Quicksort**, para ordenar dicho bloque.

Una vez ordenado, se utiliza el método **write** del objeto **FileWriter** creado previamente para añadir el bloque ordenado al archivo auxiliar número 1, separando cada elemento con una ",". Al finalizar el bloque, se escribe una diagonal "/".

Lo mismo se realiza para el siguiente bloque de **n** claves, colocando los números en el ArrayList **numerosDouble2**, ordenando dicho bloque y escribiéndolo en el segundo archivo auxiliar separando los elementos con comas y marcando el final del bloque con "/".

Luego, añadimos las claves que no fueron añadidas a los bloques, ordenamos de nueva cuenta dichas claves sobrantes y las colocamos en el archivo correspondiente (1 o 2) dependiendo de en qué archivo terminó la escritura de los bloques de **n** claves. Con esto, logramos no perder ningún número independientemente del tamaño de las particiones que se quieran realizar sobre el archivo original.

En caso de que se deseara realizar el ordenamiento de manera descendente, el método **fase1BloquesDescendente** realizaría lo mismo que el método ascendente con la única diferencia de que el **Quicksort** utilizado será una modificación que permite ordenar de manera descendente. Todo lo demás en el método se realiza de la misma manera que en su versión ascendente. Por lo tanto, hasta el momento se ha logrado leer el archivo original, separar las claves obtenidas en bloques de **n** claves, ordenar cada uno de los bloques (de manera ascendente o descendente), escribir los bloques ordenados intercalando entre el archivo auxiliar 1 y el archivo auxiliar 2 y escribir las claves sobrantes en los archivos auxiliares según se necesite.

Posterior a la separación en bloques y ordenamiento, se pasa a la segunda fase del ordenamiento por polifase. la cual consiste en un ciclo **do-while** de lecturas y escrituras de los bloques de claves hasta que se deje de cumplir la condición "verificaOrdenamiento() == false". El primer método utilizado durante la segunda fase del ordenamiento por polifase es **lecturaArchivo**. En él, se pasa como parámetro el nombre del archivo a leer, se lee una cierta cantidad de claves hasta llegar a una "/" (fin de un bloque), el cual es almacenado en el arreglo **bloque**. Luego, se separan las claves por "," y se almacenan en un arreglo auxiliar **bloqueAux** para finalmente ser convertidos a Double y almacenados en el ArrayList **bloqueArchivoString**, el cual es devuelto por la función.

Otro método útil para la segunda fase del ordenamiento por polifase es **impresionBloques**. En este método, se pasa el nombre del archivo donde proviene el bloque y se imprime el ArrayList con las claves utilizando un ciclo **for-each**.

Más adelante, el método **escrituraBloquesArchivoOriginalyTercerArchivoAscendente** creará dos nuevos objetos de tipo **FileWriter** para escribir sobre el archivo original y el tercer archivo auxiliar. Este método realiza la escritura de los bloques ordenados previamente alternando el archivo en donde se escribirán dichos bloques.

Sin embargo, la manera en que se unen los bloques previamente ordenados para escribirlos en los archivos original y auxiliar 3 es diferente de lo que se había realizado previamente, por ejemplo, en mezcla equilibrada. En este caso, el bloque leído del primer archivo auxiliar se añade al ArrayList **numerosDouble** y ocurre lo mismo con el bloque leído del segundo archivo auxiliar, el cual también es añadido a **numerosDouble**. La diferencia es que en lugar de mezclar o intercalar ambos bloques utilizando alguna especie de **merge**, se vuelve a ordenar el bloque que ahora contiene al primer bloque del archivo auxiliar uno y dos utilizando **Quicksort** para después colocarlo ya ordenado en el archivo original o tercer archivo auxiliar.

El proceso anterior es repetido hasta que se termine de vaciar los bloques del archivo auxiliar 1 y 2, obteniendo así bloques ordenados más grandes de lo que se tenía, pero ahora ubicados en el archivo original y tercer archivo auxiliar, recordando que con cada final de bloque se escribe una diagonal "/" para que la siguiente lectura del archivo pueda determinar el final de los bloques.

Lo mismo ocurre para el ordenamiento de tipo descendente. Sin embargo, los bloques unidos del archivo auxiliar 1 y del archivo auxiliar 2 ahora se intercalan utilizando una versión descendente de **Quicksort**.

Luego, todavía dentro de la segunda fase de ordenamiento por polifase, se llama al método **escrituraBloquesAuxiliaresAscendente** o al método **escrituraBloquesAuxiliaresDescendente**, según sea el caso. En cualquiera de estos métodos, se crean dos objetos de tipo **FileWriter** para escritura en los archivos auxiliares 1 y 2. Lo que se hace en este caso es similar a lo que sucedía en **escrituraBloquesArchivoOriginalyTercerArchivoAscendente**, pero ahora se escriben los bloques intercalados del archivo original y del tercer archivo auxiliar en los archivos auxiliares 1 y 2.

El proceso mencionado anteriormente se repite hasta que el método **verificaOrdenamiento** devuelva el valor **true**. Este método cuenta las diagonales "/" en el archivo original al momento de leerse y, dependiendo del número de diagonales, regresa un valor **boolean** que avisa si el archivo se encuentra o no ordenado. En este caso, se cuentan las diagonales del archivo original y del tercer archivo auxiliar y, en caso de que haya una o ninguna en el original y ninguna en el tercer archivo auxiliar, se devuelve **true**.

A continuación, se presenta una imagen con los métodos que son llamados durante la fase dos del ordenamiento por polifase (ascendente):

```
public void fase2OrdenamientoAscendente(){
    do{
        bloqueArchivo1String = lecturaArchivo(nomAux1);
        impresionBloques("1",bloqueArchivo1String);
        bloqueArchivo2String = lecturaArchivo(nomAux2);
        impresionBloques("2",bloqueArchivo2String);
        escrituraBloquesArchivoOriginalyTercerArchivoAscendente();
        bloqueArchivo3String = lecturaArchivo(nomAux3);
        if(!bloqueArchivo3String.isEmpty()){
            bloqueArchivo3String.clear();
            bloqueArchivoOriginalString = lecturaArchivo(archivoOriginal);
            impresionBloques("Original",bloqueArchivoOriginalString);
            bloqueArchivo3String = lecturaArchivo(nomAux3);
            impresionBloques("3",bloqueArchivo3String);
            escrituraBloquesAuxiliaresAscendente();
        }
    }while(verificaOrdenamiento()==false);
}
```

Esto tomando en cuenta que en la primera fase se tuvo que haber dividido en bloques de **n** claves, ordenado los bloques e intercalado dichos bloques en los archivos auxiliares 1 y 2.

No obstante, el método llamado desde el **main** del programa es **ordenamiento**. En este método, dependiendo del tipo de ordenamiento deseado, se llama a la primera fase del ordenamiento por polifase, luego a la segunda para, finalmente, leer el archivo original y realizar la impresión de los bloques utilizados. Tal como puede observarse a continuación:

```
public void ordenamiento(){
    if(tipoOrdenamiento==1){
        fase1BloquesAscendente();
        fase2OrdenamientoAscendente();
        bloqueArchivoOriginalString = lecturaArchivo(archivoOriginal);
        impresionBloques("Original",bloqueArchivoOriginalString);
    }

    if(tipoOrdenamiento == 2){
        fase1BloquesDescendente();
        fase2OrdenamientoDescendente();
        bloqueArchivoOriginalString = lecturaArchivo(archivoOriginal);
        impresionBloques("Original",bloqueArchivoOriginalString);
    }
}
```

Distribución.

La distribución es un algoritmo externo de ordenamiento en el cual se van analizando cada uno de los dígitos de los números que deseamos ordenar, para que, dependiendo del valor del dígito evaluado, el número sea almacenado en algún archivo o estructura auxiliar, y posteriormente, estos números sean recabados de cada una de las estructuras para volver a ser analizados, pero esta vez, en la siguiente posición del dígito del número a evaluar.

Dentro de la clase de **Distribución**, se realizó un procedimiento para poder realizar este ordenamiento externo a partir de un archivo original que contiene el conjunto de números a ordenar y de un conjunto de 10 archivos auxiliares, que nos permitieron ir almacenando los números dependiendo del valor del dígito que era evaluado en cada caso.

Lo primero que se realiza al elegir el ordenamiento externo por distribución es pedirle al usuario el nombre del archivo de donde se obtendrán los números a ordenar y el tipo de ordenamiento, ya sea ascendente o descendente. Dentro del **main**, se van a almacenar estos datos y se va a crear una instancia de la clase **Distribucion**, en la cual, se cuenta con un método constructor el cual al recibir los datos del nombre del archivo y el tipo de ordenamiento, los va a almacenar en atributos de la instancia, siendo estos los atributos con nombre **nombreArchivoOriginal** y **tipoOrdenamiento**. Luego, dentro del **main** se puede observar que a partir de la instancia que se creó, se manda llamar al método **ordenamiento**.

Dentro del método **ordenamiento**, se hace uso de diversos métodos, los cuales se van a explicar a medida que son utilizados, para tener una mejor idea de cómo funciona este algoritmo creado en **NetBeans** en el lenguaje de programación **Java**. Lo primero que se hace dentro de este método es mandar llamar al método **crearArchivo**, el cual es un método que no devuelve ningún valor, pero que recibe como parámetro el nombre de un archivo, junto con la extensión del mismo. Para este proyecto, los nombres de los archivos a utilizar fueron almacenados en atributos de la clase, donde cada una de las colas es representada por un atributo de tipo String, con nombre **Q#** donde el símbolo "#" es reemplazado por el dígito que representa a dicha cola, donde el atributo es inicializado y declarado con el nombre del archivo que íbamos a utilizar para almacenar las colas,

siendo estos **Q#.txt**, donde nuevamente, el # es reemplazado por el dígito que corresponde a cada cola. Dentro del método **crearArchivo**, se hace uso de un **try-catch**, el cual nos permite a la vez hacer uso de un **FileWriter** con el cual se indica cada uno de los archivos que deseamos utilizar, el **FileWriter** cuenta con la característica de que en caso de que no existe el archivo lo crea, por lo tanto, en esta función solamente se indicó cada uno de los archivos que deseamos utilizar o "crear", y por medio del **FileWriter**, se crean y en caso de existir, gracias al uso del booleano false, se va a eliminar todos los datos que contenían dichos archivos.

Después de haber creado todos los archivos auxiliares referidos al manejo de colas o a la impresión del procedimiento o de las iteraciones dentro del ordenamiento, se hace uso de una nueva función llamada **lecturaOriginal**, el cual es un método que no devuelve ningún elemento y que no recibe ningún tipo de parámetro, sino que trabaja con los atributos que tenemos dentro de la clase. Este método sirve para leer el archivo original, el cual fue indicado por el usuario dentro del **main** del proyecto. Se guardan todos los números contenidos dentro de dicho archivo original dentro de un arreglo **números**, donde cada número es almacenado dentro de una posición del arreglo gracias a que se utiliza a la "," como un indicador de separación entre números. Luego, se hace uso de un **ArrayList** de tipo **Double**, con el cual se almacenan los elementos contenidos dentro del arreglo a un **ArrayList** que los contiene como números y, finalmente, estos números contenidos dentro del **ArrayList** son almacenados en un **LinkedList** de tipo **String** y se borran todos los elementos contenidos dentro del archivo original.

Como siguiente método a utilizar sería el método **ValorRecursivoMáximo**, el cual es un método que no devuelve ningún valor y no recibe ningún parámetro, sin embargo, se encarga de recorrer la lista ligada que contiene a todos los elementos de tipo **String** y busca cual es la longitud máxima de estos elementos, para que, de esta forma, se tenga un control sobre cuál es el número que cuenta con una mayor cantidad de dígitos para que al momento de realizar el ordenamiento y la revisión de los dígitos de cada número, se verifique verdaderamente la posición de cada uno de los dígitos de todos los números de forma uniforme. Este valor lo almacenará en el atributo**ValorRecursivoMaximo**.

Ahora se hace uso del método **ArregloDeArreglos**, que sigue siendo un método que no devuelve ningún valor y no recibe ningún parámetro, pero que trabaja con los atributos de la clase. Dentro de este método, se va a hacer uso de un **ArrayList** que contiene **ArrayList** de tipo **Double**. Esta estructura es inicializada y lo primero que se hace es almacenar todos los números contenidos dentro de nuestro ArrayList de Double a la primera posición de cada uno de los **ArrayList** contenidos dentro de nuestro arreglo de arreglos, para que, de este modo, se tengan todos los números almacenados dentro de este **ArrayList**. Posteriormente, se va a descomponer a cada uno de los números en dígitos, donde cada dígito va a ocupar una posición dentro de nuestro arreglo de arreglos de **Doubles**, para el caso de trabajar con los puntos decimales, se hace uso de su clave en código **ASCII**, para que, en caso de que se encuentre un punto, éste lo trabaje como un cero. Así mismo, se busca uniformizar el número total de decimales para que, en caso de que el número no contara con punto o no contara con decimales, el método le completara los decimales que le faltaban con dígitos "cero"; del mismo modo, en caso de que el número evaluado fuera muy corto o pequeño, se iba a completar los dígitos que le faltaban mediante el uso de dígitos "cero". Después de estos procedimientos, dentro del arreglo de arreglos de valores **Double**, se tendría un conjunto de arreglos de misma longitud que tendrían sus números en la primera posición de los arreglos contenidos y que se encontrarían uniformizados con la misma cantidad de números decimales y la misma cantidad de dígitos para su evaluación dentro de funciones posteriores.

Finalmente, dentro del método **ordenamiento**, se va a empezar a realizar la evaluación de cada uno de los dígitos contenidos dentro de los números para realizar el ordenamiento, para esto, se hace uso de un ciclo **for**, el cual va a evaluar el último dígito de cada número hasta el primer dígito de cada número, haciendo uso del atributo **ValorRecursivoMaximo**, que nos indicaba la longitud máxima de los números a ordenar, lo cual lo logramos a partir de la uniformización de números en el método **ArregloDeArreglos**. Mediante el método de **radixSort**, no se devuelve ningún valor, pero se recibe como parámetro la posición de los números que se está evaluando, por lo que, se va a trabajar con la estructura arreglo de arreglos que creamos con anterioridad donde descompusimos todos los números en sus dígitos y dependiendo del valor de los dígitos, se iba a escribir el número original en cada una de las colas o archivos auxiliares que utilizamos hasta que ya no se contara con más números por distribuir o para clasificar por medio de los dígitos contenidos dentro de la posición verificada.

Ahora, dependiendo del tipo de ordenamiento ingresado por el usuario, el cual es evaluado por un if, se va a hacer uso de los métodos **pasarAOriginalAscendente** o **pasarAOriginalDescendente**, donde estos métodos hacen uso de dos funciones que se van a explicar a continuación.

El primer método del que hacen uso es del método **lecturaArchivo**, el cual devuelve un **ArrayList** de tipo **String** y recibe como parámetros el nombre de un archivo y un identificador de este archivo, para que, dentro de este método se lea el archivo indicado por el usuario y se almacenen los números dentro de un arreglo, donde cada uno de los números serán divididos tomando como indicador la separación de las comas y almacenando los números en posiciones individuales. Los números contenidos son mostrados en pantalla y son escritos en un archivo auxiliar que luego nos permitirá realizar un análisis de todas las iteraciones y el proceso de ordenamiento del archivo original. Se devuelve el **ArrayList** que contiene a todos los números almacenados dentro del archivo.

Como segundo método, se hace uso del método **vaciarColas**, el cual no devuelve ningún valor, sin embargo, recibe como parámetro un **ArrayList** de tipo **String**, el mismo que fue recuperado de la función explicada con anterioridad, para que, de este **ArrayList**, se vayan sacando de la cola cada uno de los elementos almacenados y estos sean escritos en el archivo original mediante un método **write(** de la clase **FileWriter**. Dentro de este método se debe verificar que el archivo original se sobreescriba para que no se pierda ningún número escrito previamente por otras colas.

Después de haber explicado estos dos métodos nos regresamos al análisis de las funciones **pasarAOriginalAscendente** o **pasarAOriginalDescendente**, donde lo primero que se realiza es leer todas las colas (sin importar el orden), para almacenarlas dentro de **ArrayList** de tipo **String** y tener todos los números almacenados en estas estructuras y finalmente, dependiendo del ordenamiento ascendente o descendente, se van a vaciar las colas al archivo original mediante el método **vaciarColas**. Si se realiza un ordenamiento ascendente se vaciarán las colas empezando por la cola cero, siguiente en la cola uno y así sucesivamente hasta llegar a la cola nueve; en caso de ser un ordenamiento descendente, se empezará a vaciar por la cola nueve, se seguirá con la cola ocho hasta llegar a la cola cero.

Finalmente, mediante un **if** se va a evaluar si no nos encontramos evaluando la primera posición de cada uno de los dígitos, de no ser así, se va a leer el archivo original mediante el método **lecturaOriginal**, dentro del cual, nuevamente se guardan todos los números contenidos dentro

de dicho archivo original dentro de un arreglo números, donde cada número es almacenado dentro de una posición del arreglo gracias a que se utiliza a la "," como un indicador de separación entre números y se hará uso de un **ArrayList** de tipo **Double**, con el cual se almacenan los elementos contenidos dentro del arreglo a un **ArrayList** que los contiene como números que se encuentran a un paso menos de encontrarse completamente ordenados. Finalmente, se borran todos los elementos contenidos dentro del archivo original.

Nuevamente, mediante el uso del método **ArregloDeArreglos**, se va a descomponer a cada uno de los números en dígitos, y se volverá a realizar las verificaciones explicadas con anterioridad para uniformizar el largo de los números y el número de decimales que contienen.

Tras todo este proceso, y gracias al ciclo **for** que realiza los métodos de **radixSort**, **pasarAOriginalAscendente** o **pasarAOriginalDescendente**, **lecturaOriginal** y **ArregloDeArreglos**, se van a ir evaluando cada una de las posiciones de cada uno de los dígitos de los números contenidos dentro del archivo original y serán ordenados según las indicaciones del usuario, al sacarlos y meterlos dentro de las diferentes colas para que, tras cada iteración, se encuentren un poco más ordenados para finalmente, tener un orden final y poder visualizar todo el procedimiento realizando en el archivo **IteracionesDistribucion.txt**.