**Análisis de mezcla equilibrada y polifase:**

**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 segundo 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:



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.

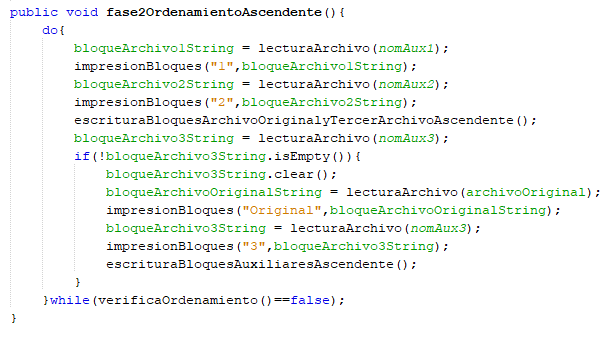
Sin embargo, la manera en que se unen los bloques previamente ordenados para escribirlos en los archivos original y auxiliar 3 es diferente de lo que se había realizado previamente, por ejemplo, en mezcla equilibrada. En este caso, el bloque leído del primer archivo auxiliar se añade al ArrayList **numerosDouble** 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):

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:

