



Universidad Nacional Autónoma de México
Facultad de Ingeniería
División de Ingeniería Eléctrica Electrónica



Proyecto 1
Implementación de algoritmos de ordenamiento
externo: Polifase, Mezcla Directa y Radix

Integrantes

Monsalvo Bolaños Melissa Monserrat
Cárdenas Cárdenas Jorge
Garrido Sánchez Samuel Arturo

Perteneciente a la asignatura de
Estructuras de datos y algoritmos II

Asignatura impartida por
M.I. Tista García Egdar

Equipo: 5
Grupo 5
Semestre 2019 - 1

Ciudad Universitaria, Coyoacán, Ciudad de México

Introducción

El ordenamiento de sistemas son útiles desde la antigüedad dado que nuestro razonamiento lógico radica en encontrar y utilizar dichos objetos del sistema para nuestro fin, sin embargo para que esto suceda de la mejor manera es necesario colocar el objeto en el lugar que le corresponda.

El razonamiento de un ordenador es más mecánico por el contrario de los humanos, por lo que su principal obstáculo radica en el gasto de recursos, que por el momento, se enfoca en la memoria y en el tiempo que se ha de tardar para que sea desarrollado el ordenamiento.

Dado lo anterior, se analizan cada algoritmo dado las características de su código, llegando a una función matemática que tiene como variable dependiente el tiempo de ejecución y como independiente la cantidad de archivos que ha de recibir.

En el manejo de Big Data es esencial contar con algún orden para hallar elementos dentro los ficheros y es un previo al manejo de bases de datos como con SQL.

Muchos de los algoritmos de ordenamiento externos se basan en dividir el trabajo en varios sub procesos, por lo que el archivo original se ve modificado y se divide en partes ya sean iguales o diferentes dependiendo el algoritmo utilizado.

Los algoritmos de ordenamiento vistos en clase corresponden al ordenamiento polifásico, mezcla directa y una interpretación externa del conocido ordenamiento Radix. Los elementos a ordenar corresponden a listas o arreglos de números ya que involucrar demás caracteres supondría utilizar una jerarquía como el código ASCII utilizado en las terminales de los ordenadores MS-Dos para el llamado de caracteres específicos.

Por último se busca que los estudiantes aprendan acerca de la importancia de utilizar éstos métodos para empezar con el manejo de archivos en el lenguaje de programación JAVA y dándole un enfoque orientado a objetos con la interfaz gráfica que se ha implementado para poder realizar el ordenamiento de una mejor manera para la experiencia de usuario.

1. Naturaleza del proyecto

1.1 Antecedentes

El ordenamiento de archivos es primordial para realizar muchos otros algoritmos como la búsqueda de manera eficiente. Existen muchos tipos de ordenamiento y empezando por su principal separación: ordenamiento interno u ordenamiento externo. El ordenamiento interno se basa en lograr ordenar un elemento que se encuentre en la memoria principal, el ordenamiento externo por contraparte se refiere a ordenar elementos dentro de un fichero de la memoria secundaria mejor conocido como almacenamiento en disco del ordenador. El segundo es a lo que se enfoca esta implementación con los algoritmos de tipo polifase, mezcla equilibrada y radix como anfitriones de la realización de todos los procesos correspondientes como la creación de archivos auxiliares para lograr simular el verdadero proceso del ordenamiento de un archivo de grandes dimensiones, haciendo particiones del total de elementos que hay y utilizando los archivos auxiliares para que así pueda ser procesador de la mejor manera

1.2 Planteamiento del problema

Se tiene un archivo de texto plano de formato .txt que se plantea como un prototipo de un enorme archivo el cual debe ser separado en secciones ya que por su tamaño es poco conveniente que la memoria se esfuerce leyendo esa cantidad de elementos uno por uno. Los métodos que se han de utilizar para desarrollar esta tarea son diversos aunque entre más veloz sea el algoritmo consume una mayor cantidad de memoria. Esto tiene un alcance en el acomodo de grandes archivos donde los elementos de éste deban ordenarse para poder realizar operaciones sobre el archivo ordenado, logrando implementar una búsqueda más eficiente que una búsqueda lineal como la binaria o incluso implementarse una tabla hash.

1.3 Objetivo del proyecto

- Identificar los algoritmos de ordenamiento externos planteados para poder analizarlos y desarrollarlos.

1.4 Actividades

- Elaborar un algoritmo que implemente el acomodo por el método POLIFASE y demostrar su funcionamiento
- Elaborar un algoritmo que implemente el acomodo por el método MEZCLA EQUILIBRADA y demostrar su funcionamiento
- Elaborar un algoritmo que implemente el acomodo por el método RADIX y demostrar su funcionamiento.
- Enfocar el trabajo a una programación orientada a objetos con uso de interfaz gráfica.

2. Marco teórico

2.1 Ordenamiento Polifásico.

El ordenamiento polifásico se basa principalmente en el uso de un número asignado por el usuario para crear listas de ese respectivo tamaño e ir mezclando internamente entre esas listas hasta convertirse en una lista. Entre los archivos reados se van intercambiando bloques que permiten distribuirse mejor la información para que el proceso manejo de la información sea de la mejor manera la para memoria. El algoritmo polifase crea archivos auxiliares donde la información del archivo original es separada en bloques de tamaño definido por el usuario y los bloques se van acomodando en los archivos auxiliares de manera que cada sub bloque sea acomodado por algún algoritmo de ordenamiento interno, en nuestro caso se utiliza QuickSort para realizar ésta tarea.

En el caso de tener un bloque con una cantidad menor a lo establecido este se limitará al último elemento dentro del archivo original donde el mini bloque será integrado al final hasta que los bloques de tamaño fijo se hayan mezclado entre ellos al combinarse entre los archivos.

2.2 Ordenamiento por Mezcla Equilibrada.

El método de ordenación por mezcla equilibrada, conocido también como natural, es una optimización del método de mezcla directa.

La idea central de este algoritmo consiste en realizar las particiones tomando secuencias ordenadas de máxima longitud en lugar de secuencias de tamaño fijo previamente determinadas.

Luego se realiza la fusión de las secuencias ordenadas, en alternada, sobre dos archivos. Aplicando estas acciones en forma repetida se logrará el archivo original quede ordenado.

El algortimo aprovecha el hecho de que entre los elementos de la secuencia original, algunos elementos consecutivos ya se encontrarán ordenados entre sí.

La mezcla natural se basa en la combinación de subsecuencias ordenadas, las cuales se distribuyen en dos cintas destino auxiliares a y b. Seguidamente se mezcla una

subsecuencia ordenada de cada cinta auxiliar. Cada pase en la mezcla natural consta de dos fases, una de distribución y otra de mezcla.

En el peor caso el número de movimientos es de orden $n \log n$, e inferior en el caso promedio.

El número de comparaciones es mucho mayor, pero al ser el coste de una comparación muy inferior al de un movimiento, este incremento no resulta significativo.

2.3 Ordenamiento por Radix

Es un algoritmo de ordenamiento estable que considera a las claves como secuencias de datos con diferentes valores posibles. Así, para ordenar las claves, el método las ordena por cada uno de sus datos, para números considera los dígitos y para las cadenas considera cada carácter. A cada ordenación de las claves por un dato se le llama iteración.

Hay dos tipos de clasificación de Radix:

MSD Radix Sort comienza a ordenar desde el dígito más significativo (el dígito que se encuentra más a la izquierda).

LSD Radix Sort comienza a ordenar desde el dígito menos significativo (el dígito que se encuentra más a la derecha)..

En este caso se analizará LSD Radix Sort externo con secuencias de dígitos.

Para realizar Radix Sort dada una lista de elementos enteros. Primero se crean los archivos auxiliares donde se almacenarán los elementos evaluados.

Posteriormente cada elemento es evaluado con respecto al dígito menos significativo.

En base a esté, el elemento es enviado al archivo correspondiente.

Una vez colocados todos los elementos dentro de los archivos, se vacía todo el contenido de cada archivo, comenzando con los que almacenan el dígito de menor valor. Cada elemento sale en el orden en el que fue ingresado.

Este procedimiento se repite hasta que cada dígito de los elementos sea evaluado.

3 Desarrollo del proyecto

3.1 Ordenamiento por método Polifase

El ordenamiento polifase desarrollado ha sido la versión establecida en clase donde las iteraciones entre F0, F2, F2 y F3 se hacen presentes. En esencia se realizan una separación de los elementos encontrados en el archivo y para un mejor manejo de los mismo se implementa una lista de listas donde la lista corresponde

Especificaciones

Respecto a los archivos:

- La extensión de los archivos debe ser txt.
- Los archivos auxiliares fueron nombrados F0, F1, F2, F3, F1-1, F2-2, Final por lo que el archivo de ingreso debe ser diferente de éstos nombres.

Respecto a los elementos:

- Los elementos no deben estar separados por una coma, solo por un espacio ya que el método que se utilizó para la lectura radica en NextInt()
- No debe contener caracteres especiales, solo números.
- Los datos deben de encontrarse en la misma línea.
-

Pseudocódigo

Mientras existan datos en doc

Leer las m llaves

Realizar sublistas de dependiendo de la cantidad m que nos haya proporcionado el usuario

Ordenar las sublistas por algún método interno

Acomodar las sublistas generadas dependiendo su paridad en 2 archivos auxiliares F1 y F2 y quedándose F1 con la lista de tamaño diferente.

Escribir los archivos F1 y F2 con los elementos de sublistas ordenadas

Mientras existan datos en F1 y F2

Realizar sublistas de tamaño m que nos hayan dado proporcionado el usuario

Ordenar las sublistas por algún método interno

Acomodar las sublistas generadas dependiendo su paridad en 2 archivos auxiliares F0 y F3 y quedándose F0 con la lista de tamaño diferente.

Mientras existan datos en F0 y F3

Mover la lista de menor tamaño a F2

Realizar sublistas de tamaño 2m que nos hayan proporcionado proporcionado el usuario

Ordenar las sublistas por algún método interno

Acomodar las sublistas generadas dependiendo su paridad en 2 archivos auxiliares F0 y F2 y quedándose F2 con la lista de tamaño diferente.

Mientras existan datos en F0 y F2

Integrar ambas listas y ordenarlas por algún métodos de ordenamiento interno

Análisis general del algoritmo

El algoritmo generar se basa en la separación de elementos de un archivo original donde cada elemento es agregado a una lista. Luego de esto se separa el mismo en sublistas para convertirse en una lista de listas manipulable que podemos jalar un conjunto de elementos datos sus características, en este caso la paridad de sus índices. Esas pequeñas sublistas son las que son ordenadas por algún método de ordenamiento externo el cual se prefirió utilizar Quick Sort (para el caso descendente utilizamos otro método quick que solo cambia en una comparación para hacer el caso descendente. Solo es necesario dentro de los parámetros insertar un String que diga descendente pero esto se encarga la interfaz gráfica con el JComboBox). Luego de esto, en la primera escritura mandamos a la lista de listas mandar las listas pares o impares a otra lista donde tendremos

Métodos importantes dentro de la clase

Sort(): Método principal que se encarga de darle vida a la ejecución. Es el que en la interfaz gráfica mandamos a llamar para empezar el acomodamiento y la escritura de los archivos auxiliares. Dentro de él podemos encontrar las funciones que se encargan de separar los bloques, ordenarlos de alguna manera interna y luego juntarlos dependiendo de su paridad. En el penúltimo movimiento cabe recalcar que m se reemplaza por $2m$ para obtener una sublista de mayor tamaño y finalmente acomodar la lista de tamaño distinto de m con la sublista mayor ordenada.

AgregarAList(): éste método se encarga de agregar los elementos que lea de un archivo de texto plano a una lista de números. Las características de éste radican en tener una función de lectura de elementos separados por espacio y no por comas. Al finalizar de acomodar los números en el arreglo retorna la lista

ImprimirListaInt(): Se encarga de imprimir la lista de números que reciba como parámetros. Su funcionamiento es simple ya que solo recorre la lista y en cada recorrido va imprimiendo el elemento en el que se encuentra.

crearSub() Dentro de éste método podremos encontrar que se asignará a una nueva lista los que se encuentren en la lista original. Se utiliza una lista de listas para poder realizar el movimiento de los bloques. De la lista original ingresada como parámetros

se harán pequeñas sublistas de tamaño m que serán insertadas dependiendo de su paridad a otra lista. Este método además de separar a la lista original en subbloques los junta en una lista de listas.

crearSub2() Tiene el mismo funcionamiento que **crearSub()** solo que esta ocasión creamos la lista de listas de 2 archivos leídos donde el proceso consiste en primero sacar todo del archivo, asignarlo a una lista, separarla en bloques en este método para crear una lista de listas con cada lista original extraída de los archivos f_1 , f_2 f_3 o dependiendo dónde nos encontremos. Cada sublista es ordenada a través de un método de ordenamiento Quick que dependerá del atributo tipo = “Ascendente” o “Descendente” el orden del ordenamiento.

MoverATxt(): Dentro de la clase existen métodos similares pero todos siguen la misma mecánica solo que algún detalle o archivo cambiado. El procedimiento radica en tener la lista de listas, separarlas dependiendo de su paridad y agregar sus elementos de las pares a una nueva lista, cabe recalcar que antes en crear sub los sub arreglos se encuentran ordenado internamente por lo que así se insertarán a las listas auxiliares f_0, f_1, f_2, f_3 de manera que se vea paso por paso el sentido del ordenamiento polifásico. Si existe una sublista de tamaño menor a m que ha quedado en el olvido se debe agregar con este método: `r.addAll(lis.subList(lis.size()-lis.size()%m, lis.size()))`; La pequeña lista corresponde en índices al tamaño de la lista menos el módulo de la lista con respecto a m . Como no sabemos si existe tal lista (en el caso que sea exacta la división entonces por eso colocamos este condicional, si es una lista vacía no la agregamos). Y llega hasta el tamaño de la lista. Con esto garantizamos que integramos la pequeña lista a continuar en la ejecución. Para finalizar a f_i y f_j los agregamos en 2 archivos distintos para que puedan ser utilizados nuevamente.

Nota importante del uso de métodos de escritura/lectura : Si usamos sólo `FileInputStream`, `FileOutputStream`, `FileReader` o `FileWriter`, cada vez que hagamos una lectura o escritura, se hará físicamente en el disco duro. Si escribimos o leemos pocos caracteres cada vez, el proceso se hace costoso y lento, con muchos accesos a disco duro. Los `BufferedReader`, `BufferedInputStream`, `BufferedWriter` y `BufferedOutputStream` añaden un buffer intermedio. Cuando leamos o escribamos, esta clase controlará los accesos a disco.

ImpresionListaInt(): Un método de impresión clásico donde se va recorriendo la lista y se imprimen los elementos que contenga

ImpresionListaDeListas(): Similiar a una impresión de listas regular solo que en esta ocasión para cada sublista mandamos a llamar al método de impresión de lista para imprimir cada sublista.

Orden: Se emplea una función orden que recibe como parámetros a la lista y un String que ese string corresponde al tipo de orden que se le dará a la lista, ascendente o descendente según lo seleccionado en la interfaz gráfica.

Constructor: Ya que nos encontramos dentro de un entorno orientado a objetos se obtuvo por realizar un método constructor ya acceder a los métodos anteriores a través de un objeto creado en la interfaz gráfica

3.2 Ordenamiento por método de Mezcla Equilibrada

Análisis general del algoritmo

La implementación del algoritmo antes descrito se realizó en el lenguaje de programación Java, y consta de dos clases que definen todos los métodos necesarios para lograr el ordenamiento bajo dicho paradigma; tales clases se describen a continuación:

Métodos importantes dentro de la clase Operaciones Archivos

Clase declarada como pública, que define un objeto a partir del cual se podrán manipular (leer y escribir) los archivos

✓ Atributos:

File file: Se define un objeto tipo File que contendrá los atributos y métodos definidos por la clase java.io.File sobre el archivo que se está manejando.

✓ Constructor:

OperacionesArchivos(File file)": este constructor recibe forzosamente para un Objeto tipo File, el cual corresponde al archivo sobre el que se ejecutaran los métodos; dicho constructor permitirá ejecutar los métodos establecidos en la clase.

✓ Métodos:

- public java.lang.String readFile(int line): Este método declarado como público recibe como argumento un entero denominado "line"; tal método lee una línea especificada del archivo, devolviendo dicha línea como String que contiene los

números y caracteres encontrados en el archivo de texto. Para ello se emplea un ciclo for que se itera desde cero hasta el número de línea deseada (line) -1, se coloca el menos uno, dado que la última línea (línea deseada), se debe establecer fuera del for para retornarla directamente, evitando con ello la creación de variables extras. Dado que este método que trabaja con archivos, arroja una `java.io.IOException`, para el caso donde no se pudo completar la lectura, o algo salió mal al intentar realizarla.

- `public void writeFile(java.lang.String cadena):` Este método es declarado como público, y recibe como argumentos un String, el cual se denomina cadena, la cual puede incluir saltos de línea y retornos de carro, los cuales deben incluirse dentro de una mima llamada a la función para optimizar el tiempo de escritura.

Dentro del método se establecen las instrucciones necesarias para la escritura en el archivo de texto, donde “cadena”, corresponde al valor a escribir sobre él; de igual forma que el método anterior, “writeFile” arroja una, por lo que es necesario declararlo dentro de un bloque try catch.

Dado que en todo algoritmo es fundamental considerar el tiempo que se tarda en realizar determinada tarea, las lecturas y escrituras que se hagan sobre los archivos se deben hacer en lo menor posible, dado que hacerlas es aún más tardado que análisis u operaciones que se hacen con los datos en si, por lo tal motivo es que las lecturas se hacen de toda una línea completa de datos hasta encontrar un retorno de carro, o salto de línea, y no de manera parcial(número por número); de manera análoga el número de veces que se desee escribir sobre el archivo debe estar en función de que tan necesaria es dicha tarea; en otras palabras, es necesario reducir en lo mayor posible el número de llamadas de los métodos para asegurar que el tiempo que la implementación del algoritmo, toma en ordenar el archivo, sea mínima.

Métodos importantes dentro de la clase Mezcla Equilibrada

En esta clase se definen todos los métodos necesarios para la realizar el ordenamiento por mezcla equilibrada de manera tanto ascendente como descendente; se compone de los elementos a continuación descritos:

✓ Atributos:

- File fileSort: Objeto de tipo File, que contendrá los métodos y propiedades establecido por su clase constructora para el manejo eficiente del archivo de texto a ordenar.
- String aux1, aux2, cadena: Conjunto de variables que contendrán las cadenas que se vayan leyendo de cada uno de los archivos necesarios en el algoritmo
- int m: entero que permitirá saber el número de iteración correspondiente del algoritmo

✓ Constructor:

El constructor que se establece para esta clase requiere que se le envíen como parámetros un objeto tipo File, y un booleano, que le permita saber la ubicación del archivo a ordenar, y el tipo de búsqueda a implementar respectivamente, true para ordenamiento Ascendente y false para ordenamiento Descendente.

Dentro del mismo se valida que los archivos aux1 y aux2 no existan, en caso no cumplirse los elimina; esto se hace con la finalidad de evitar colisiones entre estos archivos.

Es importante destacar al momento de crear el objeto, se realiza el ordenamiento, es decir, cuando se invoca el método constructor, éste manda a llamar a los métodos respectivos para el ordenamiento de manera automática.

✓ Métodos:

La implementación del ordenamiento de Mezcla Equilibrada, utiliza tres objetos de tipo OperacionesArchivos, que corresponderán a: archivo original (archivo a ordenar), archivo aux1 y archivo aux2 (los cuales contendrán las particiones de cada iteración), de tal manera que mediante dichos objetos sea posible leer y escribir tales archivos.

Mediante un ciclo While, el algoritmo se repite hasta que se llega a un caso base, el cual corresponde a la lista (contenida en el archivo) ordenada, donde se rompe el ciclo, y se informa al usuario que su archivo ha sido ordenado.

La implementación se hace siguiendo la lógica del algoritmo, es decir, respetando los pasos esenciales que caracterizan al algoritmo, siendo estos: la creación de particiones del archivo externo de manera equilibrada, de forma que los elementos

que componen cada partición se encuentren ya ordenados bajo el criterio establecido (ascendente o descendente), dichas particiones se debe repartir en los dos archivos auxiliares, de tal manera que en el primer archivo queden las particiones impares, y en segundo las pares; con dichas particiones se hace una mezcla, tomando una del archivo aux1 y otra del archivo aux2, estas se comparan entre sí, y una vez comparadas todas las particiones, estas se escriben sobre el archivo original, de manera tal que es posible observar cada una de las iteraciones y cambios que se realizan al aplicar el algoritmo sobre un archivo de texto, cabe destacar que el paso anterior es meramente ilustrativo y con fines didácticos de la aplicación del algoritmo.

La implementación creada del algoritmo también valida que el archivo no se encuentre vacío, donde en caso de cumplirse dicha aseveración, se le informa al usuario que el archivo no contiene elemento y se rompe con el ciclo, evitando así salidas inesperadas, tales como un error en la lectura de datos nulos.

Las clases generadas se listan a continuación:

- MezclarAscentede()ordena un archivo con bajo un criterio de orden ascendente
- mezclaDescendente(): ordena un archivo bajo un criterio descendente, las particiones cumplen con tal criterio
- mixNumbersA(java.util.ArrayList<java.lang.String> arr1, java.util.ArrayList<java.lang.String> arr2): este método recibe como argumentos dos ArrayList, y se encarga de realizar la comparación y ordenamiento entre las particiones del archivo en cada iteración correspondiente de manera Ascendente.
- mixNumbersD(java.util.ArrayList<java.lang.String> arr1, java.util.ArrayList<java.lang.String> arr2): este método es idénticamente igual al anteriormente descrito, a diferencia que en él, la mezcla de las particiones genera elementos ordenados de manera descendente.
- add1Comas():Añade una separador (",") a la variable String aux1
- add2Comas(): Añade una separador (",") a la variable String aux2

3.3 Ordenamiento por método Radix

Pseudocódigo

Radix_Sort(lista):

Para cada posición de una clave
 Para cada elemento de la lista
 Toma el componente de la clave en la posición actual
 Introduce el elemento en el archivo correspondiente
 Une todos los elementos de los archivos

Análisis general del algoritmo

La clase Radix Sort tiene como atributos una lista de archivos, donde se almacenan los archivos auxiliares y un archivo que copia el contenido del archivo original.

El método radix

Es el primer método que se debe ejecutar. Es un método público que recibe como argumento el nombre del archivo original, posteriormente a través de un switch podemos mostrar al usuario un menú, donde tiene tres opciones, ordenar la lista de manera descendente, ascendente o salir. En caso de que elija la primer opción, primero se borran todos los archivos auxiliares, si es que existen, posteriormente se crea una copia del archivo original, esto para poder observar las iteraciones cuando se juntan todos los elementos de los archivos auxiliares. Imprime la lista original y luego llama al método radixArchivo, a partir de este se realiza el ordenamiento. Al terminar se copia la lista ya ordenada al archivo original y se imprime nuevamente.

Si el usuario ingresa la segunda opción se realizan los pasos anteriores, con la única diferencia de que se llama al método radixInverso, el cual ordenará la lista de forma descendente.

Si el usuario elije la tercera opción, sale del while y termina la ejecución del programa.

Método radixArchivo

Es el método principal, recibe como argumento una cadena con el nombre del archivo que contiene la lista a ordenar. Primero inicializa la lista de archivos, insertando 10 archivos, uno para cada dígito del 0 al 9. Posteriormente recupera la lista de elementos del archivo copia y calcula el total de dígitos del número mayor, esto nos indicará el número de iteraciones que se realizarán. Inicializa un arreglo de boléanos para el control de cambio de archivos.

Para cada elemento de la lista, calcula un índice, el cual corresponde al dígito menos significativo.

Para ello ocupamos esta expresión:

$$\begin{aligned} \text{clave} &= (a/\text{div}); \\ \text{clave} &= \text{clave}\%10; \end{aligned}$$

Donde a representa el elemento a ordenar, div va descartando cada dígito evaluado, en cada iteración se multiplica por 10; MOD recupera el siguiente dígito a evaluar.

Este índice corresponde a la posición en la lista de archivos auxiliares. Se escribe el elemento en el archivo contenido en esa posición y al mismo tiempo se marca como cambio en el archivo.

Posteriormente se unen todos los elementos de los archivos auxiliares. Para todos los archivos de la lista, si hubo cambios en los archivos, entonces se recupera la última línea del archivo con una lista de cadenas auxiliar y posteriormente se escribe cada elemento en el archivo copia. Aquí termina una iteración. Se limpian las variables auxiliares, se decrementa el contador de iteraciones, se imprimen las listas en los archivos auxiliares y la lista con la unión de todos los elementos.

Este proceso se repite tantos dígitos tenga el elemento más grande a evaluar.

Método radixInverso

Realiza el ordenamiento descendente, es prácticamente igual al anterior, solo que aquí la asignación de la clave cambia.

$$\begin{aligned} clave &= (a/div); \\ clave &= clave\%10; \\ clave &= 9 - clave; \end{aligned}$$

Donde 9 representa la última posición del arreglo de archivos auxiliares, de este modo, los números con dígitos de menor valor, se envían a los archivos que se vacían al último y por lo tanto los de mayor valor quedan al principio de la lista.

Método leerl

Este método recibe como argumento una cadena con el nombre del archivo a leer, crea un objeto file, el cual se relaciona al archivo copia. Filereader nos permite leer el archivo y BufferedReader nos permite almacenar lo recuperado.

Cada renglón del archivo es almacenado en una lista de cadenas donde la última cadena almacenada será la que se recupere, a partir de esta cadena y mediante el método Split se recupera un arreglo de todos los elementos separados por coma y los almacena en un arreglo. Posteriormente se realiza un casteo a enteros y se almacenan en una lista. Posteriormente cerramos los canales de datos. El método regresa la lista de elementos enteros.

Método leer

Realiza las mismas operaciones que el anterior con la única diferencia de que no realiza un casteo a enteros, regresa la lista de cadenas.

Método escrele

Con este método podemos escribir sobre un archivo sin utilizar saltos de línea, sirve para escribir elementos. Recibe como argumento la cadena con el nombre del archivo en el que se escribirá y la cadena que se escribirá. Se crea un objeto file y se asocia con el archivo. FileWriter tiene como función escribir datos en un archivo, BufferedWriter reserva un espacio en memoria donde se almacena la información escrita en un archivo y PrintWriter escribe directamente sobre el archivo.

A través de printwriter invocamos el método write y le ingresamos la cadena que deseamos escribir. Posteriormente cerramos los canales de datos.

Todo este proceso debe realizarse dentro de un try, si hay un error nos manda un mensaje con el problema.

Método inicializar

Crea 10 archivos auxiliares, uno por cada dígito y los almacena dentro de una lista.

Método mayor

Recibe una lista de enteros en la cual realiza una búsqueda lineal del elemento más grande, comparando cada elemento de la lista. Al encontrar el número más grande, realiza un casteo para utilizar el método length, el cual nos dará el número de dígitos que compone dicho elemento que a su vez corresponde al número de iteraciones del RadixSort. Devuelve el entero obtenido.

Método imprimir

Recibe una cadena que corresponde el nombre del archivo a imprimir, este así vez llama al método leerl y almacena en una lista auxiliar los elementos enteros recuperados, mediante un ciclo for los va imprimiendo.

Método eliminar

Recibe una cadena con el nombre del archivo a eliminar, si existe el archivo lo elimina con el método delete.

Método eliminartodo

Elimina todos los archivos auxiliares. Si la lista de archivos auxiliares no está vacía, elimina cada archivo de la lista llamando al método eliminar, por ultimo elimina el archivo copia.

Método copia

Genera una copia del archivo original. Recibe como cadenas el nombre del archivo original y el nombre del archivo donde se realiza la copia de contenido. Crea una lista de Strings que recupera a través del método lee todos los elementos de la última línea del archivo a copiar en forma de cadenas y los escribe en el archivo receptor.

3.4 Interfa gráfica

La interfaz gráfica fue generada gracias a elementos de tipo JFrame de netBeans. Los elementos a recalcar son los siguientes:

```
jSplitPanel = new javax.swing.JSplitPane();
jSplitPanel.setBackground(Color.white);
jLabel1 = new javax.swing.JLabel();
jSeparator1 = new javax.swing.JSeparator();
jLabel2 = new javax.swing.JLabel();
cmbMetodo = new javax.swing.JComboBox<>();
jLabel3 = new javax.swing.JLabel();
cmbCampo = new javax.swing.JComboBox<>();
abrirArchivo = new javax.swing.JButton();
ruta_archivo = new javax.swing.JLabel();
jSeparator2 = new javax.swing.JSeparator();
btnOrdenar = new javax.swing.JButton();
```

JSplitPane hace referencia a las pequeñas separaciones que contiene el programa, en este caso fue solo inicializado con un init

setBackground configura el fondo de una sección para que sea blanco mediante el parámetro

JLabel inserta una caja de texto en el frame de manera que pueda contener texto para mostrar, es además utilizado para mostrar la dirección del archivo seleccionado

JComboBox corresponde al seleccionador por ejemplo como el del Método u Orden.

Para poder saber lo seleccionado es necesario utilizar el método:

elJComboBox.getSelectedItem().toString() y nos devuelve un String con su contenido.

JBoton: Integra botones a nuestro programa. Al presionar un botón se realiza una acción. Requiere un objeto de tipo evento para funcionar pero dentro de su contenido podemos ingresar la función o el método que debe realizar que en nuestro caso es un tipo seleccionado con los parámetros dados por la ruta_archivo y los ComboBox.

Layout: Se basa en la integrar todos los elementos dichos anteriormente pero asignarles ubicaciones dentro de la ventana, tamaño e incluso características especiales.

Abrir Archivo: De igual manera se emplea un botón que realiza una acción que corresponde a lo siguiente:

Crea un objeto de tipo fileChooser que corresponde a

```
openFileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);  
fileChooser.addChoosableFileFilter(new FileNameExtensionFilter("*.txt", "txt"));
```

```
openFileChooser.setMultiSelectionEnabled(false);  
int seleccion = openFileChooser.showOpenDialog(null);
```

En éstas funciones primero se despliega una venta para poder seleccionar un archivo, no se puede seleccionar múltiple.

Esto es para seleccionar un TXT fuera de la carpeta del Proyecto pero lo hace una copia en la carpeta para hacer notar los cambios de orden.

```
try{  
    Path origenPath =  
    FileSystems.getDefault().getPath(file.toString());  
    Path destinoPath =  
    FileSystems.getDefault().getPath(new File(".").getCanonicalPath()  
    + "/" + file.getName());  
    System.out.println(destinoPath);  
    Files.copy(origenPath, destinoPath,  
    StandardCopyOption.REPLACE_EXISTING);  
}
```

4 Conclusión

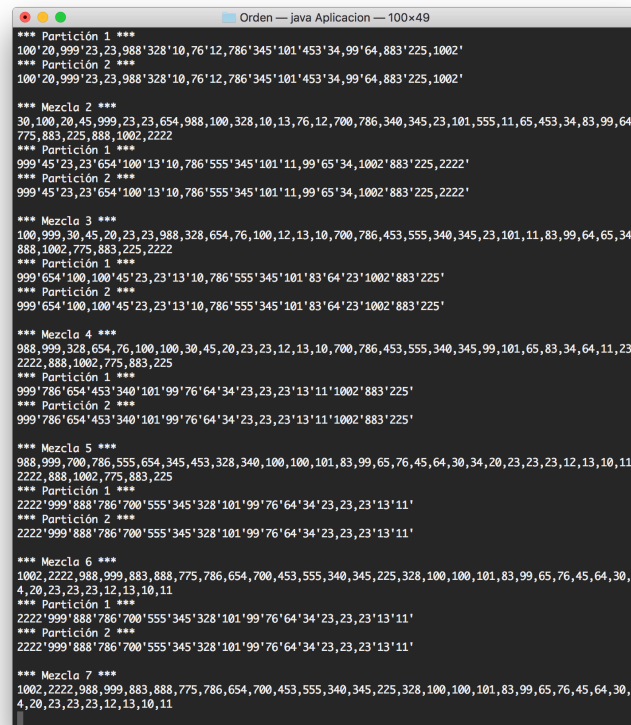
Los algoritmos de ordenamiento externo son muy útiles para implementar orden en grandes archivos donde la cantidad de elementos es tal que es necesario separarlo por partes para poder procesar todos con mejor disposición. Existen muchos tipos de ordenamientos pero algunos cuentan con mejores características que otros que entra la parte de memoria-tiempo. El orden de los archivos es esencial para realizar otras tareas como lo puede ser la búsqueda como principal ejemplo donde un archivo ordenado y especialmente con ese tamaño pueda encontrarse un elemento con mayor facilidad que ir recorriendo todo el elemento hasta encontrar coincidencias como es el caso de la búsqueda binaria. Estos procesos desarrollados se llevan a cabo en el desarrollo de bases de datos y es una introducción para el manejo de las mismas. Para esta practica se cumplieron con los objetivos planteados en los requerimientos del proyecto además de aprender a integrar nuevas funciones a nuestros programas.

5 Bibliografías

<https://stackoverflow.com/questions/15771949/how-do-i-make-jfilechooser-only-accept-txt>
https://issuu.com/jairomartiner/docs/teor_a_de_algoritmos_word
<https://www.redeszone.net/2012/02/20/curso-de-java-entrada-y-salida-con-ficheros-clase-scanner/>
<https://stackoverflow.com/questions/17383867/set-icon-image-in-java>
<https://www.growingwiththeweb.com/2014/05/counting-sort.html>
<https://upcommons.upc.edu/bitstream/handle/2117/93297/05Djg05de14.pdf?sequence=5&isAllowed=y>
<https://revistas.ufpr.br/letras/article/download/51234/33743>
<https://www.cs.helsinki.fi/u/tpkarkka/opetus/10s/spa/lecture06.pdf>
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.7021&rep=rep1&type=pdf>

6 Anexos

Capturas de pantalla



```
*** Partición 1 ***
100'20,999'23,23,988'328'10,76'12,786'345'101'453'34,99'64,883'225,1002'
*** Partición 2 ***
100'20,999'23,23,988'328'10,76'12,786'345'101'453'34,99'64,883'225,1002'

*** Mezcla 2 ***
30,100,20,45,999,23,23,654,988,100,328,10,13,76,12,700,786,340,345,23,101,555,11,65,453,34,83,99,64,
775,883,225,888,1002,2222
*** Partición 1 ***
999'45'23,23'654'100'13'10,786'555'345'101'11,99'65'34,1002'883'225,2222'
*** Partición 2 ***
999'45'23,23'654'100'13'10,786'555'345'101'11,99'65'34,1002'883'225,2222'

*** Mezcla 3 ***
100,999,30,45,20,23,23,988,328,654,76,100,12,13,10,700,786,453,555,340,345,23,101,11,83,99,64,65,34,
888,1002,775,883,225,2222
*** Partición 1 ***
999'654'100,100'45'23,23'13'10,786'555'345'101'83'64'23'1002'883'225'
*** Partición 2 ***
999'654'100,100'45'23,23'13'10,786'555'345'101'83'64'23'1002'883'225'

*** Mezcla 4 ***
988,999,328,654,76,100,100,30,45,20,23,12,13,10,700,786,453,555,340,345,99,101,65,83,34,64,11,23,
2222,888,1002,775,883,225
*** Partición 1 ***
999'786'654'453'340'101'99'76'64'34'23,23'13'11'1002'883'225'
*** Partición 2 ***
999'786'654'453'340'101'99'76'64'34'23,23'13'11'1002'883'225'

*** Mezcla 5 ***
988,999,700,786,555,654,345,453,328,340,100,100,101,83,99,65,76,45,64,30,34,20,23,23,12,13,10,11,
2222,888,1002,775,883,225
*** Partición 1 ***
2222'999'888'786'700'555'345'328'101'99'76'64'34'23,23'13'11'
*** Partición 2 ***
2222'999'888'786'700'555'345'328'101'99'76'64'34'23,23'13'11'

*** Mezcla 6 ***
1002,2222,988,999,883,888,775,786,654,700,453,555,340,345,225,328,100,100,101,83,99,65,76,45,64,30,3
4,20,23,23,12,13,10,11
*** Partición 1 ***
2222'999'888'786'700'555'345'328'101'99'76'64'34'23,23'13'11'
*** Partición 2 ***
2222'999'888'786'700'555'345'328'101'99'76'64'34'23,23'13'11'

*** Mezcla 7 ***
1002,2222,988,999,883,888,775,786,654,700,453,555,340,345,225,328,100,100,101,83,99,65,76,45,64,30,3
4,20,23,23,12,13,10,11
```

MezclaEquilibrada

```
Orden — java Aplicacion — 100x49
Lista original:
2222,1002,999,988,888,883,786,775,700,654,555,453,345,340,328,225,101,100,100,99,83,76,65,64,45,34,3
0,23,23,23,20,13,12,11,10,
iterción*****
File0.txt: 999,99,
File1.txt: 988,888,328,
File3.txt: 786,76,
File4.txt: 775,555,345,225,65,45,
File5.txt: 654,64,34,
File6.txt: 883,453,83,23,23,23,13,
File7.txt: 2222,1002,12,
File8.txt: 101,11,
File9.txt: 700,340,100,100,30,20,10,

File10.txt: 999,99,988,888,328,786,76,775,555,345,225,65,45,654,64,34,883,453,83,23,23,23,13,2222,10
02,12,101,11,700,340,100,100,30,20,10,
iterción*****
File0.txt: 999,99,
File1.txt: 988,888,786,883,83,
File2.txt: 76,775,
File3.txt: 65,64,
File4.txt: 555,654,453,
File5.txt: 345,45,340,
File6.txt: 34,30,
File7.txt: 328,225,23,23,23,2222,20,
File8.txt: 13,12,11,10,
File9.txt: 1002,101,700,100,100,

File10.txt: 999,99,988,888,786,883,83,76,775,65,64,555,654,453,345,45,340,34,30,328,225,23,23,23,222
2,20,13,12,11,10,1002,101,700,100,100,
iterción*****
File0.txt: 999,988,
File1.txt: 888,883,
File2.txt: 786,775,700,
File3.txt: 654,
File4.txt: 555,
File5.txt: 453,
File6.txt: 345,340,328,
File7.txt: 225,2222,
File8.txt: 101,100,100,
File9.txt: 99,83,76,65,64,45,34,30,23,23,23,20,13,12,11,10,1002,

File10.txt: 999,988,888,883,786,775,700,654,555,453,345,340,328,225,2222,101,100,100,99,83,76,65,64,
45,34,30,23,23,23,20,13,12,11,10,1002,
iterción*****
File7.txt: 2222,
File8.txt: 1002,
File9.txt: 999,988,888,883,786,775,700,654,555,453,345,340,328,225,101,100,100,99,83,76,65,64,45,34,
30,23,23,20,13,12,11,10,
```

Radix Sort

```
Orden — java Aplicacion — 81x48
Lista: 100 20 30 45 23 999 23 100 988 654 328 10 12 76 56 53 45 13 54

Lista: 100 20 30 45

Lista: 23 999 23 100

Lista: 988 654 328 10

Lista: 12 76 56 53

[20 30 45 100 ]
[23 23 100 999 ]
[10 328 654 988 ]
[12 53 56 76 ]
[13 45 54 ]

- - - - -
Lista: 20 30 45 100

Lista: 10 328 654 988

[20 30 45 100 ]
[10 328 654 988 ]
- - - - -
Lista: 23 23 100 999

Lista: 12 53 56 76

[23 23 100 999 ]
[12 53 56 76 ]
- - - - -
Lista: 20 23 23 30 45 100 100 999 13 45 54

Lista: 10 12 53 56 76 328 654 988

- - - - -
Lista: 20 23 23 30 45 100 100 999

[20 23 23 30 45 100 100 999 ]
Lista: 10 12 53 56 76 328 654 988

[10 12 53 56 76 328 654 988 ]
LISTA SEMIFINAL
Lista: 10 12 20 23 23 30 45 53 56 76 100 100 328 654 988 999

Lista: 13 45 54

F1-1: 10 12 20 23 23 30 45 53 56 76 100 100 328 654 988 999
```

Polifase

Interfaz Gráfica

