INSTITÚTO TECNOLÓGICO DE MEXICO

CAMPUS ORIZABA

NOMBRE DE LA ASIGNATURA:
ESTRUCTURA DE DATOS

TEMA:
REPORTE DE PRACTICA TEMA 5 MÉTODOS DE ORDENAMIENTOS

NOMBRE DE LOS INTEGRANTES:
HERNÁNDEZ DÍAZ ARIEL - 21010195
PELACIOS MENDEZ XIMENA MONSERRAT – 21010209
VELAZQUEZ SARMIENTO CELESTE - 21010223
ESPINDOLA OLIVERA PALOMA - 21010186

NOMBRE DE LA PROFESORA:
MARIA JACINTA MARTINEZ CASTILLO

HORARIO OFICIAL DE LA CLASE:
15:00 – 16:00 HRS

PERIODO:
ENERO - JUNIO 2023

INTRODUCCIÓN

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar. En este caso, nos servirán para ordenar vectores o matrices con valores asignados aleatoriamente. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada algoritmo. Este informe nos permitirá conocer más a fondo cada método distinto de ordenamiento, desde uno simple hasta el más complejo. Se realizarán comparaciones en tiempo de ejecución, pre-requisitos de cada algoritmo, funcionalidad, alcance, etc.

Ordenar es simplemente colocar información de una manera especial basándonos en un criterio de ordenamiento. El propósito principal de un ordenamiento es el de facilitar las búsquedas de los registros del conjunto ordenado. Un ordenamiento es conviene usarlo cuándo se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.

Es importante destacar que existen diferentes técnicas de ordenamiento como lo es; El Método Burbuja, que consiste en comparar pares de valores de llaves; Método Selección, el cual consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo; y por último el Método Intercalación, con el cual se combinan los sub-archivos ordenados en una sola ejecución.

COMPETENCIA(S) A DESARROLLAR

Comprende y aplica estructuras no lineales para la solución de problemas.

MARCO TEORICO

Burbuja con señal

Consiste en utilizar una marca o señal para indicar que no se ha producido ningún intercambio en una pasada.

Comprueba si el arreglo está totalmente ordenado después de cada pasada terminando su ejecución en caso afirmativo.

Existen tres tipos de casos:

- El mejor de los casos (Ordenado)
- El caso medio (Desordenado)
- El peor de los casos (Orden Inverso)

Este método es eficiente cuando los valores del vector se encuentran ordenados o semi ordenados. Se basa en el mismo proceso del ordenamiento burbuja simple, pero si en una pasada no ocurren intercambios quiere decir que el vector esta ordenado, por lo tanto hay que implementar un mecanismo para detener el proceso cuando comienza una pasada sin intercambios.

Doble Burbuja

Este algoritmo se puede definir como la variación del tipo burbuja . En el caso de la ordenación de burbujas, los elementos adyacentes se comparan desde el comienzo de la matriz y el elemento máximo se almacenará en la posición ordenada adecuada después de la primera iteración. El segundo elemento más grande se almacenará en su posición ordenada después de la segunda iteración y así sucesivamente y obtendremos la matriz ordenada.

Shell (incrementos - decrementos)

El método de ordenamiento Shell consiste en dividir el arreglo (o la lista de elementos) en intervalos (o bloques) de varios elementos para organizarlos después por medio del ordenamiento de inserción directa. El proceso se repite, pero con intervalos cada vez más pequeños, de tal manera que al final, el ordenamiento se haga en un intervalo de una sola posición, similar al ordenamiento por inserción directa, la diferencia entre ambos es qué, al final, en el método ShellSu nombre proviene de su creador, Donald Shell, y no tiene que ver en la forma como funciona el algoritmo. los elementos ya están casi ordenados.

Selección directa

Consiste en buscar el menor elemento del arreglo y colocarlo en la primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se coloca en la segunda posición. El proceso continua hasta que todos los elementos del arreglo han sido ordenados.

Se basa en realizar varias pasadas, intentando encontrar en cada una de ellas el elemento que según el criterio de ordenación es mínimo y colocándolo posteriormente en su sitio.

Inserción directa

El método de ordenación por inserción directa consiste en recorrer todo el array comenzando desde el segundo elemento hasta el final. Para cada elemento, se trata de colocarlo en el lugar correcto entre todos los elementos anteriores a él o sea entre los elementos a su izquierda en el array.

Dada una posición actual p, el algoritmo se basa en que los elementos A[0], A[1], ..., A[p-1] ya están ordenados.

Binaria

El método por inserción binaria es una mejora del método de inserción directa. La mejora consiste en realizar una búsqueda binaria en lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado.

El proceso al igual que el de inserción directa, se repite desde el 2do hasta el n-ésimo elemento. Toma su nombre debido a la similitud de ordenamiento de los árboles binarios. El método de ordenación por inserción binaria realiza una búsqueda binara en lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado. El proceso se repite desde el segundo hasta el n-ésimo elemento.

HeapSort

Este método se llama mezcla porque combina dos o más secuencias en una sola secuencia ordenada por medio de la selección repetida de los componentes accesibles en ese momento. Un arreglo individual puede usarse en lugar de dos secuencias si se considera como de doble extremo.

En este caso se tomarán elementos de los dos extremos del arreglo para hacer la mezcla.

El destino de los elementos combinados se cambia después de que cada par ha sido ordenado para llenar uniformemente las dos secuencias que son el destino. Después de cada pasada los dos extremos del arreglo intercambian de papel, la fuente se convierte en el nuevo destino y viceversa. La idea central de este algoritmo

consiste en la realización sucesiva de una división y una fusión que producen secuencias ordenadas de longitud cada vez mayor.

Quicksort recursivo

QuickSort es un algoritmo basado en el principio “divide y vencerás”. Es uno de los algoritmos más rápidos conocidos para ordenar. Este método es recursivo aunque existen versiones con formas iterativas que lo hacen aún más rápido. Su funcionamiento es de la siguiente manera:

- Se elige un número del vector como referencia, al que se le llamará “pivote”.
- Reordenar el arreglo de tal forma que los elementos menores al pivote queden en el lado izquierdo y al lado derecho los elementos mayores.
- Se hace uso de la recursividad para ordenar tanto el conjunto de la izquierda como el de la derecha.

Radix

Este algoritmo funciona trasladando los elementos a una cola, comenzando con el dígito menos significativo del número (El número colocado más a la derecha tomando en cuenta unidades, decenas, centenas, etc.). Cuando todos los elementos son ingresados a las colas, éstas se recorren en orden para después agregarlos al vector.

Este algoritmo es muy rápido en comparación con otros algoritmos de ordenación, sin embargo ocupa más espacio de memoria al utilizar colas.

Intercalación

En este método de ordenamiento existen 2 archivos con llaves ordenadas, las cuales se mezclan para formar un solo archivo. Se combinan los sub-archivos ordenados en una sola ejecución. La longitud de los archivos puede ser diferente. El proceso consiste en leer un registro de cada archivo y compararlos, el menor es almacenado en el archivo de resultado y el otro se compara con el siguiente elemento del archivo si existe.

El proceso se repite hasta que alguno de los archivos quede vació y los elementos del otro archivo se almacenar

Este algoritmo de comparación es ya estable que se mantiene el orden relativa a los registros con claves iguales

Mezcla Directa

El método MergeSort es un algoritmo de ordenación recursivo con un número de comparaciones entre elementos del array mínimo.

Su funcionamiento es similar al Quicksort, y está basado en la técnica divide y vencerás.

De forma resumida el funcionamiento del método MergeSort es el siguiente:

- Si la longitud del array es menor o igual a 1 entonces ya está ordenado.

- El array a ordenar se divide en dos mitades de tamaño similar.

- Cada mitad se ordena de forma recursiva aplicando el método MergeSort.

- A continuación, las dos mitades ya ordenadas se mezclan formando una secuencia ordenada.

Mezcla Natural

El método de Mezcla Natural consiste en aprovechar la existencia de secuencias ya ordenadas dentro de los datos de los archivos. A partir de las secuencias ordenadas existentes en el archivo, se obtienen particiones que se almacenan en dos archivos o ficheros auxiliares. Las particiones almacenadas en estos archivos auxiliares se fusionan posteriormente para crear secuencias ordenadas cuya longitud se incrementa arbitrariamente hasta conseguir la total ordenación de los datos contenidos en el archivo original.

RECURSOS, MATERIALES Y EQUIPO

- Computadora
- Java
- Lectura de los materiales de apoyo del tema 5 (AULA PADLET)
- Notas de clase (problemas resueltos en clase y materiales de trabajo de la profesora)

METODO DE ORDENAMIENTO BURBUJA CON SEÑAL

Análisis de eficiencia

Este método es eficiente cuando los valores del vector se encuentran ordenados o semi ordenados. Se basa en el mismo proceso del ordenamiento burbuja simple, pero si en una pasada no ocurren intercambios quiere decir que el vector esta ordenado, por lo tanto hay que implementar un mecanismo para detener el proceso cuando comienza una pasada sin intercambios.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- El mejor de los casos (Ordenado)
- El caso medio (Desordenado)

- El peor de los casos (Orden Inverso)

Complejidad en el tiempo y complejidad espacial.

METODO DE ORDENAMIENTO SHELL

Análisis de eficiencia

Este método es eficiente cuando los valores del vector se encuentran ordenados o semi ordenados. Se basa en el mismo proceso del ordenamiento burbuja simple, pero si en una pasada no ocurren intercambios quiere decir que el vector esta ordenado, por lo tanto hay que implementar un mecanismo para detener el proceso cuando comienza una pasada sin intercambios.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- El mejor de los casos (Ordenado)
- El caso medio (Desordenado)
- El peor de los casos (Orden Inverso)

Complejidad en el tiempo y complejidad espacial.

METODO DE ORDENAMIENTO HEAPSHORT

Análisis de eficiencia

Este método es eficiente cuando los valores del vector se encuentran ordenados o semi ordenados. Se basa en el mismo proceso del ordenamiento burbuja simple, pero si en una pasada no ocurren intercambios quiere decir que el vector esta ordenado, por lo tanto hay que implementar un mecanismo para detener el proceso cuando comienza una pasada sin intercambios.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- El mejor de los casos (Ordenado)
- El caso medio (Desordenado)
- El peor de los casos (Orden Inverso)

Complejidad en el tiempo y complejidad espacial.

METODO DE ORDENAMIENTO QUICKSORT RECURSIVO

Análisis de eficiencia

El método quicksort es el más rápido de ordenación interna que existe en la actualidad. Esto es sorprendente, porque el método tiene su origen en el método de intercambio directo, el peor de todos los métodos directos. Diversos estudios realizados sobre su comportamiento demuestran que si se escoge en cada pasada el elemento que ocupa la posición central del conjunto de datos a analizar, el número de pasadas necesarias para ordenarlo es del orden de log n. Respecto del número de comparaciones, si el tamaño del arreglo es una potencia de 2, en la primera pasada realizará (n-1) comparaciones. en la segunda (n-1)/2 comparaciones, pero en dos conjuntos diferentes, en la tercera realizará (n - 1)/4 comparaciones, pero en cuatro conjuntos diferentes y así sucesiva- mente.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $O(n \cdot \log n)$.
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.
- En el caso promedio, el orden es $O(n \cdot \log n)$.

METODO DE ORDENAMIENTO RADIX

Análisis de eficiencia

Depende en que las llaves estén compuestas de bits aleatorios en un orden aleatorio. Si esta condición no se cumple ocurre una fuerte degradación en el desempeño de estos métodos. Adicionalmente, requiere de espacio adicional para realizar los intercambios. Los algoritmos de ordenamiento basados en radix se consideran como de propósito particular debido a que su factibilidad depende de propiedades especiales de las llaves, en contraste con algoritmos de propósito general como Quicksort que se usan con mayor frecuencia debido a su adaptabilidad a una mayor variedad de aplicaciones.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

1. Mejor caso:
    - En el mejor caso, Radix Sort tiene una complejidad lineal O(n), donde "n" es el número de elementos a ordenar.
    - Esto ocurre cuando todos los elementos tienen el mismo número de dígitos y no hay necesidad de realizar ninguna iteración adicional.
    - Por ejemplo, si tenemos los siguientes números de tres dígitos: [123, 456, 789], en el primer paso ya estarían ordenados debido a que se comparan solo los dígitos de las unidades.

2. Caso medio:
    - En el caso medio, Radix Sort tiene una complejidad O(k * n), donde "k" es el número promedio de dígitos de los elementos a ordenar.
    - Esto ocurre cuando los elementos tienen diferentes cantidades de dígitos y se deben realizar múltiples iteraciones para ordenarlos completamente.
    - Por ejemplo, si tenemos los siguientes números: [123, 4567, 89, 12, 3456], se requerirán varias iteraciones para ordenar todos los dígitos.
3. Peor caso:
    - En el peor caso, Radix Sort tiene una complejidad O(k * n), similar al caso medio.
    - Esto ocurre cuando los elementos tienen diferentes cantidades de dígitos y se requieren muchas iteraciones para ordenarlos.
    - Por ejemplo, si tenemos los siguientes números: [987, 654, 321, 12345, 6789], se requerirán más iteraciones y comparaciones para ordenar todos los dígitos.

Complejidad en el tiempo:

- La complejidad en el tiempo de Radix Sort depende del número de dígitos necesarios para representar los elementos a ordenar y del tamaño del conjunto de datos.
- En general, la complejidad en el tiempo de Radix Sort es O(d * (n + b)), donde "d" es el número de dígitos, "n" es el número de elementos a ordenar y "b" es la base utilizada para representar los dígitos (por lo general, b = 10 para representación decimal).
- Dado que el número de dígitos "d" puede ser proporcional al logaritmo del valor máximo del conjunto de datos, la complejidad en el tiempo puede considerarse O((log(max) * (n + b)), donde "max" es el valor máximo en el conjunto de datos.

Complejidad espacial:

- La complejidad espacial de Radix Sort depende del tamaño del conjunto de datos a ordenar.
- La complejidad espacial de Radix Sort es O(n + b), donde "n" es el número de elementos a ordenar y "b" es la base utilizada para representar los dígitos.
- Esto se debe a que Radix Sort requiere almacenar temporalmente los elementos en estructuras de datos auxiliares, como colas o arreglos, para realizar las pasadas de ordenación basadas en los dígitos.

METODO DE ORDENAMIENTO SELECCION DIRECTA

Análisis de eficiencia

El **ordenamiento por selección** mejora el ordenamiento burbuja haciendo un sólo intercambio por cada pasada a través de la lista. Para hacer esto, un ordenamiento por selección busca el valor mayor a medida que hace una pasada y, después de completar la pasada, lo pone en la ubicación correcta. Al igual que con un ordenamiento burbuja, después de la primera pasada, el ítem mayor está en la ubicación correcta. Después de la segunda pasada, el siguiente mayor está en su ubicación.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- Mejor caso: O(n^2)
- Peor caso: O(n^2)
- Caso promedio: O(n^2)

Complejidad en el tiempo y complejidad espacial.

Tiempo:

El análisis del algoritmo de selección es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño de la lista o array y no de la distribución inicial de los datos.

Espacio:

Se puede observar que el uso de arreglos nativos es unas 50 veces más eficiente que la cola. Sin embargo, el algoritmo de ordenamiento por selección y reemplazo sigue siendo O(n^2) y por lo tanto es ineficiente.

METODO DE ORDENAMIENTO INSERCION DIRECTA

Análisis de eficiencia

En este método lo que se hace es tener una sublista ordenada de elementos del arreglo e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sub lista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- En el peor de los casos, el tiempo de ejecución en $O(n^2)$.
- En el mejor caso el tiempo de ejecución de este método de ordenamiento es O(n).
- El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Cuanto más ordenada esté inicialmente más se acerca a O(n) y cuanto más desordenada, más se acerca a $O(n^2)$.
- El peor caso el método de inserción directa es igual que en los métodos de burbuja y selección, pero el mejor caso podemos tener ahorros en tiempo de ejecución

Complejidad en el tiempo y complejidad espacial.

Tiempo: Caso de borde: cuando la posición del mínimo es k==i. Entonces se intercambia a[i] con a[i]. Una rápida revisión del intercambio permite apreciar que en ese caso a[i] no altera su valor, y por lo tanto no es incorrecto.

Espacio: Se podría introducir una instrucción if para que no se realice el intercambio cuando k==i, pero esto sería menos eficiente que realizar el intercambio, aunque no sirva. En efecto, en algunos casos el if permitiría ahorrar un intercambio, pero en la mayoría de los casos, el if no serviría y sería un sobrecosto que excedería con creces lo ahorrado cuando sí sirve.

METODO DE ORDENAMIENTO INSERCION BINARIA

Análisis de eficiencia

El ordenamiento binaria típica es la búsqueda de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o de la palabra que busca. Se puede tener suerte y acertar con la página correcta, pero normalmente no será así y se mueve el lector a la página anterior o posterior del libro.

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

- Caso medio

La búsqueda binaria tiene una complejidad logarítmica logn comparada con la complejidad lineal n de la búsqueda lineal utilizada en la ordenación por inserción. Utilizamos la ordenación binaria para n elementos lo que nos da la complejidad temporal nlogn. Por tanto, la complejidad temporal es del orden de [Big Theta]: O(nlogn).

- El peor caso

El peor caso se produce cuando el array está ordenado de forma inversa y se requiere el máximo número de desplazamientos. La complejidad temporal en el peor caso es del orden de [Big O]: O(nlogn).

- El mejor caso

El mejor caso se produce cuando el array ya está ordenado y no es necesario desplazar elementos. La complejidad temporal en el mejor caso es [Big Omega]: O(n).

Complejidad en el tiempo y complejidad espacial.

La complejidad espacial del algoritmo de ordenación binaria es O(n) porque no se necesita más memoria que una variable temporal.

IMPLEMENTACION DE METODOS

METODO ARRAY VACIO

```java
@Override
public boolean arrayVacio() {
    return (p==-1);
}
```

Este método nos regresara un verdadero o falso si el arreglo esta vacio, este mismo lo ocuparemos para cada método en caso de que queramos que nos muestre datos (a pesar de que no hay nada).

METODO ESPACIO ARRAY

```java
@Override
public boolean espacioArray() {
    return (p<=tam);
}
```

Este otro método, hará casi lo mismo que el anterior solo que en este nos preguntara si aun hay espacio en el arreglo, si lo hay nos dejara meter mas datos y sino entonces nos dirá que el array está lleno.

METODO VACIAR ARRAY

```java
@Override
public void vaciarArray() {
    datos = new int[tam];
    p=-1;
}
```

Este último nos indicará que hay que vaciar el arreglo, esto en caso de que queramos vaciar el arreglo y poner otros.

METODO ALMACENAR ALEATORIOS

```java
@Override
public void almacenaAleatorios() {
    for(int i =0; i<datos.length;i++){
        datos[i] = generaRandom(1, 10);
        p++;
    }
}
```

Este método almacenara los datos generados por medio del método generarRandom (el cual genera los datos aleatorios). Así mismo este incrementara para irse almacenando en el arreglo.

METODO IMPRIME DATOS

```java
@Override
public String imprimeDatos() {
    String cad="";
    for (int i = 0; i<=p; i++){
        cad+= i+"=>[" + datos[i] + "]" + "\n";
    }
    return "\n" + cad;
}
```

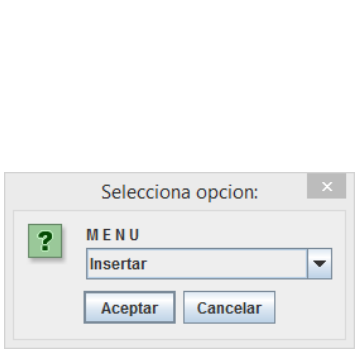
En este método, nos permitirá imprimir el arreglo ya sea en primera instancia desordenado, y en otro caso cuando llamemos a algún método de ordenamiento, y obviamente lo ordene.

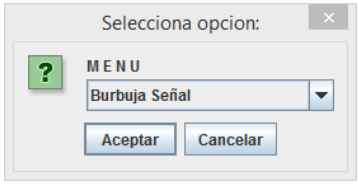
METODO BURBUJA CON SEÑAL

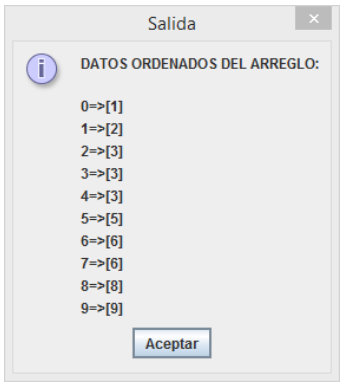
```java
@Override
public void burbujaSeñal() {
    boolean band;
    int vueltas = 0;
    for (int i = 0; i < datos.length - 1; i++) {
        band = false;
        for (int j = 0; j < datos.length - i - 1; j++) {
            if (datos[j] > datos[j + 1]) {
                int temp = datos[j];
                datos[j] = datos[j + 1];
                datos[j + 1] = temp;
                band = true;
                vueltas++;
            }
        }
        if (!band)
            break;
    }
    Tools.imprime("Vueltas: "+vueltas);
}
```

En este método llamado burbuja con señal lo que vamos a hacer es tener una variable booleana para que nos permita hacer funcionar el código, usaremos 2 arreglos

EJECUTABLES METODO BURBUJA CON SEÑAL





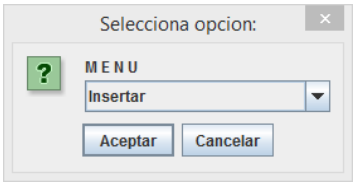


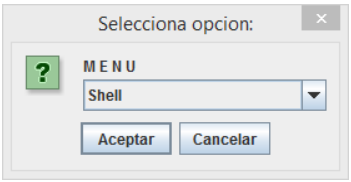
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

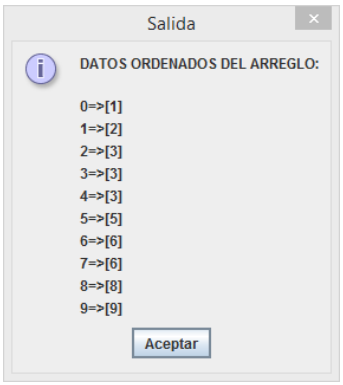
METODO SHELL INCREMENTOS-DECREMENTOS

```
@Override
public void shellIncreDecre() {
    int a = datos.length / 2;
    while (a > 0) {
        for (int i = a; i < datos.length; i++) {
            int tm = datos[i], x = i;
            while (x >= a && datos[x - a] > tm) {
                datos[x] = datos[x - a];
                x -= a;
            }
            datos[x] = tm;
        }
        a/= 2;
    }
}
```

EJECUTABLES METODO SHELL INCREMENTOS-DECREMENTOS





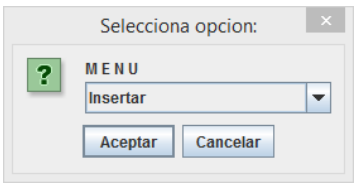


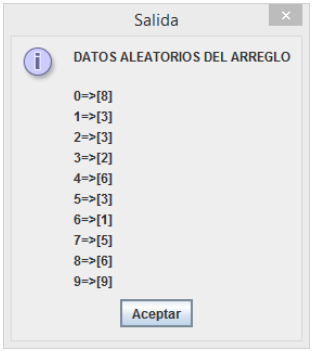
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

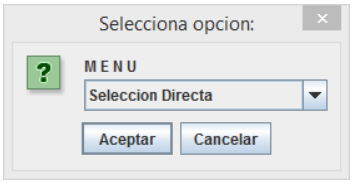
METODO SELECCIÓN DIRECTA

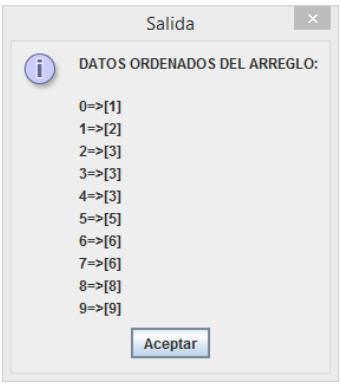
```
@Override
public void seleDirecta() {
    for (int i = 0; i < datos.length - 1; i++) {
        int Min = i;
        for (int x = i + 1; x < datos.length; x++) {
            if (datos[x] < datos[Min]) {
                Min = x;
            }
        }
        int tp = datos[Min];
        datos[Min] = datos[i];
        datos[i] = tp;
    }
}
```

EJECUTABLES METODO SELECCIÓN DIRECTA







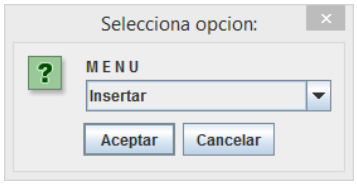


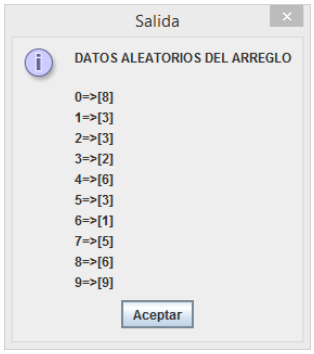
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

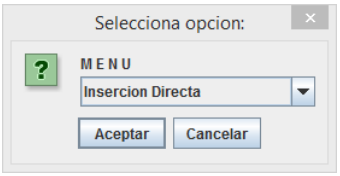
METODO INSERCION DIRECTA

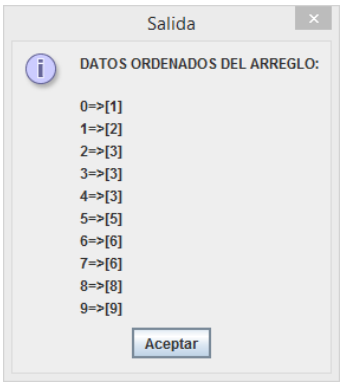
```
@Override
public void inserDirecta() {
    for (int i = 1; i < datos.length; i++) {
        int wiss = datos[i]; int x = i - 1;
        while (x >= 0 && datos[x] > wiss) {
            datos[x + 1] = datos[x];
            x--;
        }
        datos[x + 1] = wiss;
    }
}
```

EJECUTABLES METODO INSERCION DIRECTA







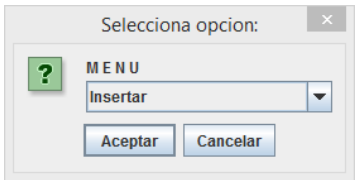


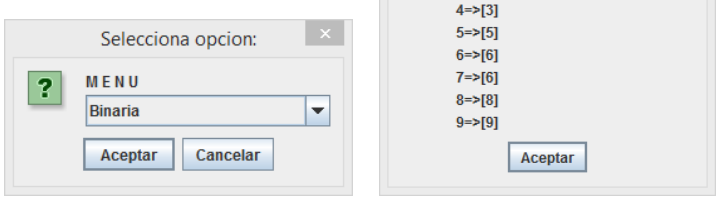
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

METODO BINARIA

```
@Override
public void binaria() {
    for (int i = 1; i < datos.length; i++) {
        int temp = datos[i],j = i - 1;
        while (j >= 0 && datos[j] > temp) {
            datos[j + 1] = datos[j];
            j--;
        }
        datos[j + 1] = temp;
    }
}
```

EJECUTABLES METODO BINARIA



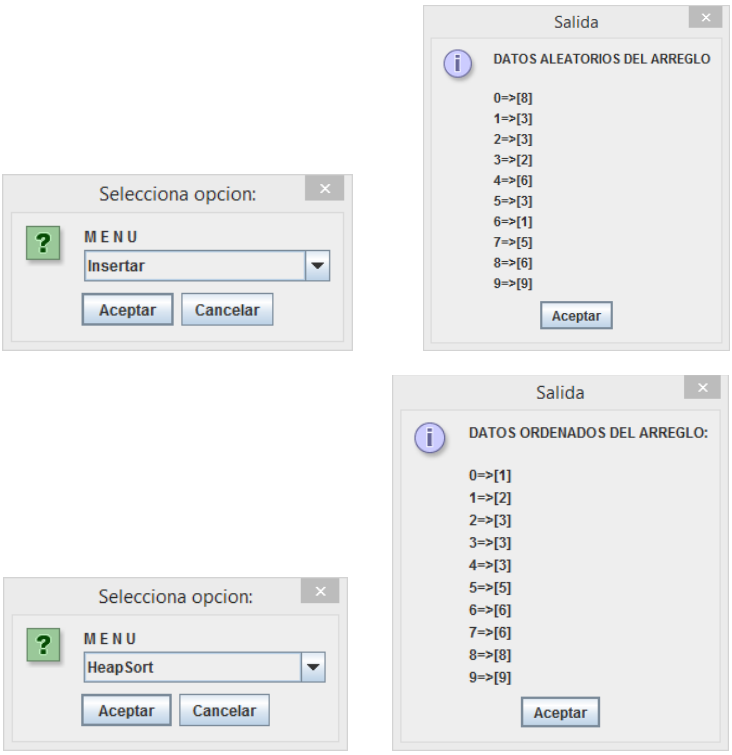


Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

METODO HEAPSORT

```
public void heapSort() {
    for (int i = datos.length / 2 - 1; i >= 0; i--) {
        heapSort2(datos, datos.length, i);
    }
    for (int i = datos.length - 1; i > 0; i--) {
        int tp = datos[0];
        datos[0] = datos[i];
        datos[i] = tp;
        heapSort2(datos, i, 0);
    }
}
public static void heapSort2(int datos[], int tamaño, int pivote) {
    int l_izq = 2 * pivote + 1;
    int l_der = 2 * pivote + 2;
    int alto = pivote;
    if (l_izq < tamaño && datos[l_izq] > datos[alto]) {
        alto = l_izq;}
    if (l_der < tamaño && datos[l_der] > datos[alto]) {
        alto = l_der;
    }
    if (alto != pivote) {
        int temp = datos[pivote];
        datos[pivote] = datos[alto];
        datos[alto] = temp;
        heapSort2(datos, tamaño, alto);}
}
```

EJECUTABLES METODO HEAPSORT



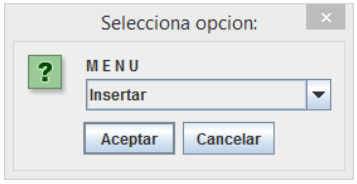
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

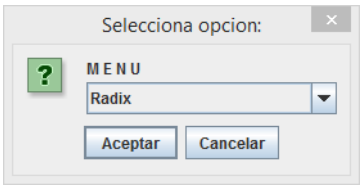
METODO RADIX

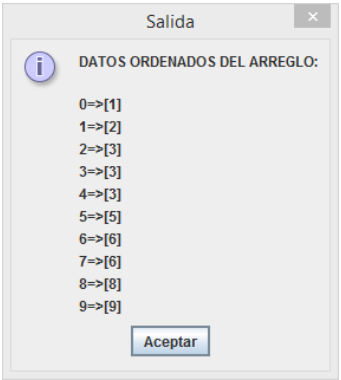
```
@Override
public void radix() {
    int may = getMax(datos);
    int oppo = 1;
    while (may / oppo > 0) {
        countSort(datos, oppo);
        oppo *= 10;
    }
}
```

```java
public static void countSort(int[] datos, int ter) {
      int n = datos.length;
      int[] output = new int[n];
      int[] count = new int[10];
      for (int num : datos) {
          count[(num / ter) % 10]++;
      }
      for (int i = 1; i < 10; i++) {
          count[i] += count[i - 1];
      }
      for (int i = n - 1; i >= 0; i--) {
          output[count[(datos[i] / ter) % 10] - 1] = datos[i];
          count[(datos[i] / ter) % 10]--;
      }
      System.arraycopy(output, 0, datos, 0, n);
  }

  public static int getMax(int[] datos) {
      int max = datos[0];
      for (int num : datos) {
          if (num > max) {
              max = num;
          }
      }
      return max;
  }
```

EJECUTABLES METODO RADIX





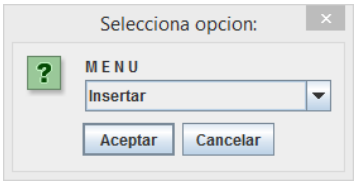


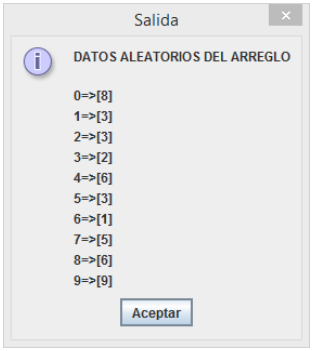
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

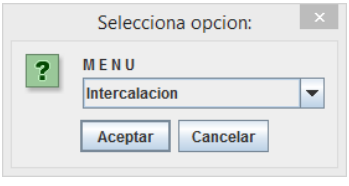
METODO INTERCALACION

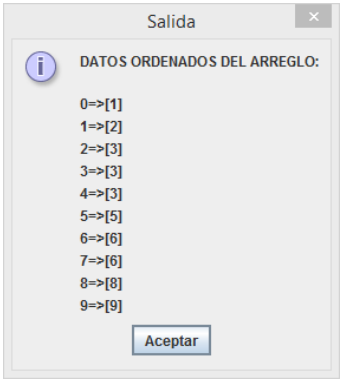
```
@Override
public void intercalacion() {
    int n = datos.length;

    if (n < 2) {
        return;
    }

    for (int i = 1; i < n; i++) {
        int elemento = datos[i];
        int j = i - 1;

        while (j >= 0 && datos[j] > elemento) {
            datos[j + 1] = datos[j];
            j--;
        }

        datos[j + 1] = elemento;
    }
}
```

EJECUTABLES METODO INTERCALACION









Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la

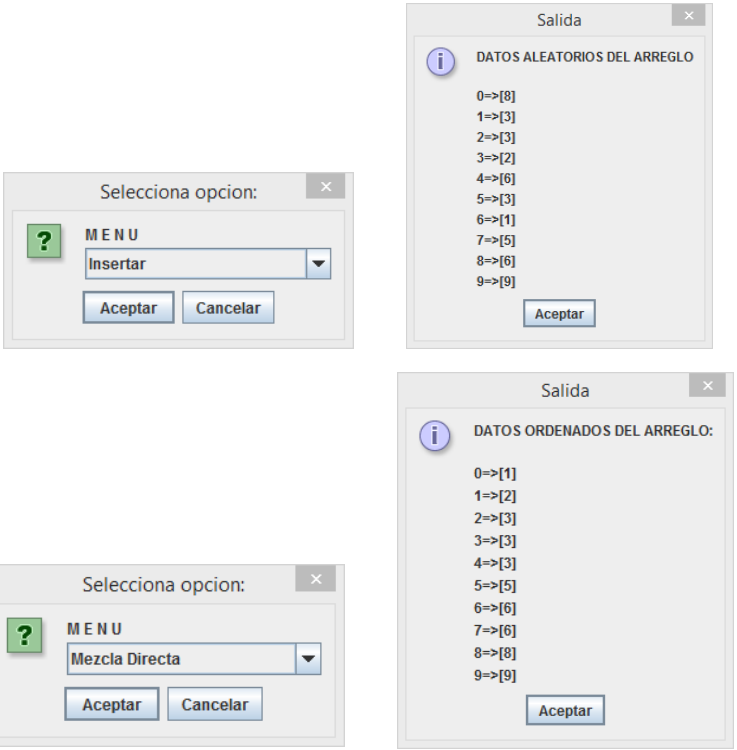
posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

METODO MEZCLA DIRECTA

```
@Override
    public void mezclaDirecta() {
        mezcla2(datos);
    }

    public static int[] mezcla2(int datos[]) {
        int i,j,k;
       if(datos.length>1) {
       int izq = datos.length/2;
       int der = datos.length - izq;
       int arregloIzq[]= new int [izq];
       int arregloDer[]= new int [der];

       for(i=0; i<izq; i++ ) {
           arregloIzq[i]=datos[i];
       }
       for(i = izq; i<izq+der;i++) {
           arregloDer[i-izq]=datos[i];
       }
       arregloIzq= mezcla2(arregloIzq);
       arregloDer=mezcla2(arregloDer);

       i = 0;
       j = 0;
       k = 0;

       while (arregloIzq.length !=j &&
arregloDer.length !=k ) {
           if (arregloIzq[j]<arregloDer[k]) {
               datos[i]=arregloIzq[j];
               i++;
               j++;
           }else {
               datos[i]=arregloDer[k];
               i++;
```

```
                k++;
            }
        }
        while (arregloIzq.length !=j) {
            datos[i]=arregloIzq[j];
            i++;
            j++;
        }
        while (arregloDer.length !=k) {
            datos[i]=arregloDer[k];
            i++;
            k++;
        }
        }
        return datos;
    }
```

EJECUTABLES METODO MEZCLA DIRECTA



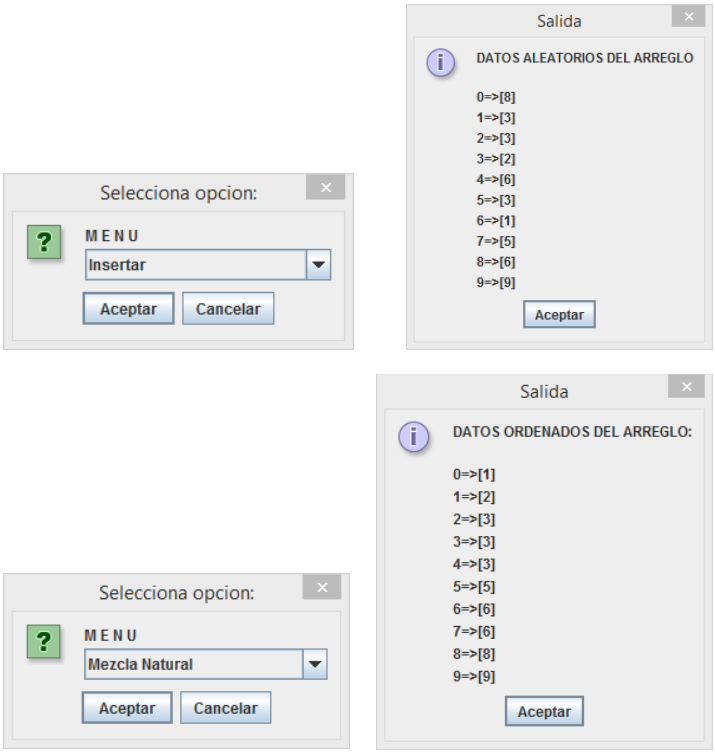
Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la

posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

METODO MEZCLA NATURAL

```
@Override
public void mezclaNatural() {
    int izquierda = 0, izq = 0, derecha = datos.length -1, der = derecha;
boolean band = false;
do {
    band = true;
    izquierda = 0;

    while (izquierda<derecha ) {
        izq = izquierda;
        while(izq < derecha && datos[izq] <= datos[izq+1]) {
            izq++;
        }
        der = izq +1;
        while(der == derecha-1 || der<derecha && datos[der]<=datos[der+1]) {
            der++;
        }
        if (der<=derecha) {
            mezcla2(datos);
            band = false;
        }
        izquierda = izq;
    }
}while(!band);

}
```


EJECUTABLES METODO MEZCLA NATURAL



Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de

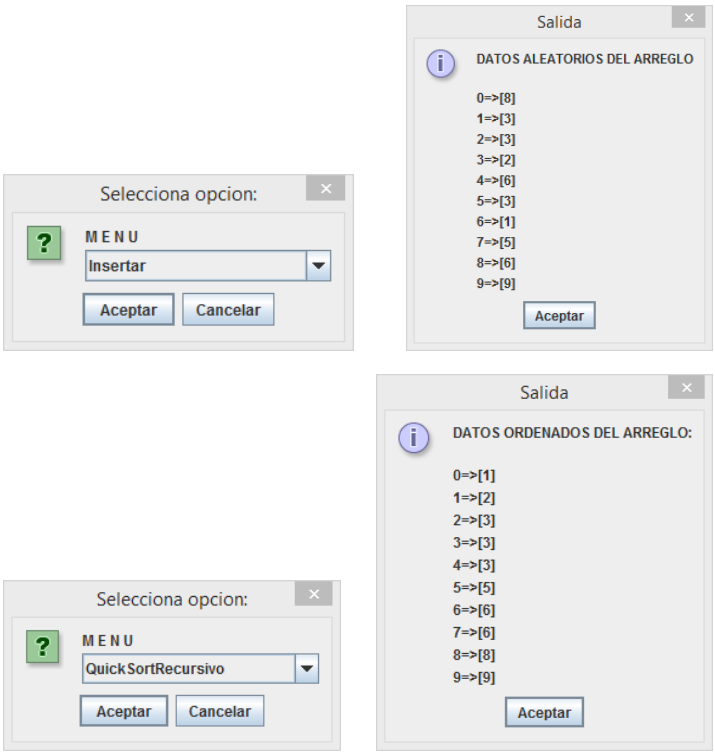
llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

METODO QUICKSORT RECURSIVO

```
@Override
    public void quicksortRecursivo() {
    quicksort2(datos, 0, datos.length - 1);
    }



    public static void quicksort2(int[] datos, int
min, int max) {
    int pivote = datos[min];

        int i = min;
        int j = max;
        int aux;

        while(i<j)
        {
           while (datos[i] <= pivote && i < j)
              i++;

           while (datos[j] > pivote)
              j--;

           if (i<j)
           {
               aux = datos[i];
               datos[i]= datos[j];
               datos[j]=aux;
           }
        }

         datos[min] = datos[j];
         datos[j] = pivote;
```

```
            if (min < j-1)
             quicksort2(datos,min,j-1);

            if (j+1 < max)
             quicksort2(datos,j+1,max);
    }
```

EJECUTABLES METODO QUICKSORT RECURSIVO



Por lo que vemos, claro está que el método si funciono correctamente, ya que como vemos en la primera pantalla el arreglo se encuentra desordenado, y después de llamar al método, este ya se encuentra ordenado. Aparte de que nos muestra la posición de cada dato. Finalmente se puede decir que el objetivo de este método si se cumplió.

METODO GENERA RANDOM

```
@Override
public int generaRandom(int min, int max) {
    return (int)((max - min + 1) * Math.random()+ min);
}
```

Este método de lo que se encarga es generar números aleatorios, y estos se almacenan agregándolos a cada método, estos serán valores del 1 al 10.

CONCLUSION

Los métodos de ordenamiento de datos son muy útiles, ya que la forma de arreglar los registros de una tabla en algún orden secuencial de acuerdo con un criterio de ordenamiento, el cual puede ser numérico, alfabético o alfanumérico, ascendente o descendente. Nos facilita las búsquedas de cantidades de registros en un moderado tiempo, en modelo de eficiencia. Mediante sus técnicas podemos colocar listas detrás de otras y luego ordenarlas, como también podemos comparar pares de valores de llaves, e intercambiarlos si no se encuentran en sus posiciones correctas.

La ordenación o clasificación es el proceso de organizar datos en algún orden o secuencia específica, tal como creciente o decreciente, para datos numéricos, o alfabéticos, para datos de caracteres. Los métodos de ordenación más directos son los que se realizan en el espacio ocupado por el array. Son de gran utilidad estos métodos ya que facilitan el trabajo de ordenamiento, cualquiera de estos programas y están diseñados para eso, por eso nos ayudan bastante en la obtención de los resultados.

BIBLIOGRAFIA

- Algoritmo burbuja con Uso DE señal. (s/f). Prezi.com. Recuperado el 29 de mayo de 2023, de https://prezi.com/icns6w2dtkgj/algoritmo-burbuja-con-uso-de-senal/?frame=54a476c817f4adb1d1d5e7b9fc0acf7c5c5487f8

- Burbuja Con Señal. (s/f). Scribd. Recuperado el 29 de mayo de 2023, de https://es.scribd.com/document/545890365/BURBUJA-CON-SENAL

- Duarte, E. (2016). MÉTODOS DE ORDENACIÓN CLASES. https://www.academia.edu/28556935/M%C3%89TODOS_DE_ORDENACI%C3%93N_CLASES

- Gorman, T. (2012, mayo 28). Estructura DE datos Tema: Método DE intercambio directo o burbuja intercambio directo con señal. SlideServe. https://www.slideserve.com/taite/estructura-de-datos-tema-m-todo-de-intercambio-directo-o-burbuja-int

- Macaray, J.-F., & Nicolas, C. (1996). Programacion java. Gestion 2000.
- Mariana, P. P. (s/f). Tips para estudiantes de Sistemas Computacionales. Blogspot.com. Recuperado el 29 de mayo de 2023, de https://tipsparaisc.blogspot.com/2010/12/ordenamiento-externo-mezcla-natural.html

- Metodo DE intercalcacion.Pdf - método DE ordenamiento Externo: IntercalaciÓn EDDJava Ordenacion externa La ordenación externa O DE archivos - CS1320. (s/f). Coursehero.com. Recuperado el 29 de mayo de 2023, de https://www.coursehero.com/file/149226703/metodo-de-intercalcacionpdf/

- Ordenación por intercambio o burbuja. (s/f). Alciro.org. Recuperado el 29 de mayo de 2023, de http://www.alciro.org/alciro/Programacion-cpp-Builder_12/ordenacion-intercambio-burbuja_447.htm

- Wikipedia contributors. (s/f). Ordenamiento de burbuja bidireccional. Wikipedia, The Free Encyclopedia. https://es.wikipedia.org/w/index.php?title=Ordenamiento_de_burbuja_bidireccional&oldid=122350722

- (S/f). Dyndns.org. Recuperado el 29 de mayo de 2023, de http://ual.dyndns.org/biblioteca/Estructura%20de%20Datos/Pdf/08%20Metodos%20de%20ordenacion.pdf