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)

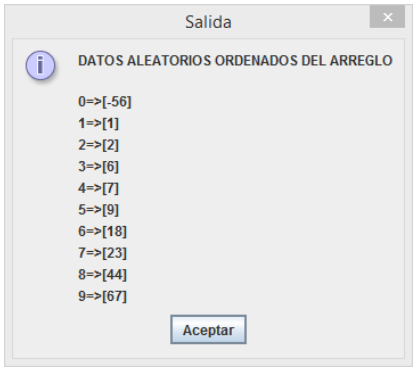
- El peor de los casos (Orden Inverso)

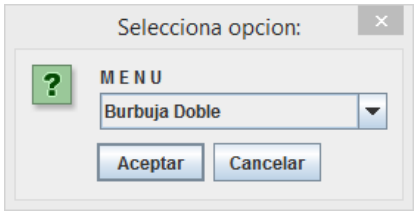
Complejidad en el tiempo y complejidad espacial.

Complejidad en el tiempo: la complejidad en el tiempo es de O(n)

Complejidad en el espacio: la complejidad en el espacio es de O(n^2)

Prueba de escritorio:





METODO DE ORDENAMIENTO DOBLE BURBUJA CON SEÑAL

Análisis de eficiencia

Es una variante del algoritmo de burbuja que utiliza una señal para mejorar la eficiencia. En este enfoque, se realizan dos pasadas: una de izquierda a derecha y otra de derecha a izquierda, y se utiliza una señal para determinar si se ha producido algún intercambio en cada pasada. Si no hay intercambios en una pasada, se asume que la lista está ordenada y se detiene el algoritmo.

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

- Mejor caso: El mejor caso ocurre cuando la lista de entrada ya está completamente ordenada. En este caso, el algoritmo solo realizará una pasada de izquierda a derecha sin realizar ningún intercambio. Luego, en la pasada de derecha a izquierda, tampoco se realizarán intercambios porque la lista ya está ordenada. Como resultado, el algoritmo se detendrá después de dos pasadas sin tener que realizar ningún intercambio adicional. La complejidad en el mejor caso es O(n), donde "n" es el número de elementos en la lista.
- Caso medio: En el caso medio, la eficiencia del método de ordenamiento "doble burbuja con señal" es similar a la del algoritmo de burbuja básico. En promedio, requerirá O(n^2) comparaciones e intercambios para ordenar la lista, donde "n" es el número de elementos. La señal solo reduce el número

de iteraciones en el mejor caso, pero no afecta significativamente el rendimiento en el caso medio.

- Peor caso: El peor caso ocurre cuando la lista de entrada está ordenada en orden descendente. En este caso, el algoritmo realizará intercambios en cada pasada, tanto de izquierda a derecha como de derecha a izquierda. Esto requerirá múltiples iteraciones hasta que la lista esté completamente ordenada. La complejidad en el peor caso es O(n^2), similar al algoritmo de burbuja básico.

Complejidad en el tiempo y complejidad espacial.

Complejidad en el tiempo:

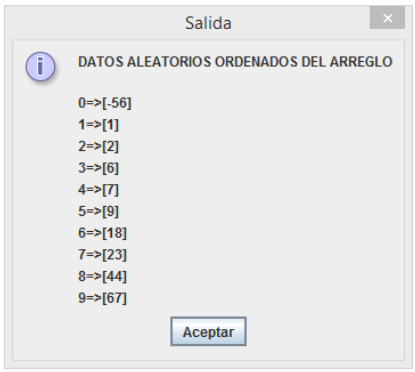
- En el mejor caso, donde la lista de entrada ya está ordenada, el método de ordenamiento de doble burbuja con señal realizará solo una pasada de izquierda a derecha y otra de derecha a izquierda sin realizar intercambios adicionales. Esto implica que el tiempo de ejecución es lineal y tiene una complejidad de O(n), donde "n" es el número de elementos en la lista.
- En el peor caso, donde la lista de entrada está ordenada en orden descendente, el método de ordenamiento de doble burbuja con señal realizará múltiples pasadas de izquierda a derecha y de derecha a izquierda hasta que la lista esté completamente ordenada. Esto resulta en una complejidad cuadrática de O(n^2), ya que se deben realizar comparaciones e intercambios para cada par de elementos en la lista.
- En el caso medio, la complejidad en el tiempo del método de ordenamiento de doble burbuja con señal también es cuadrática, O(n^2), ya que no hay optimizaciones significativas en comparación con el algoritmo de burbuja básico.

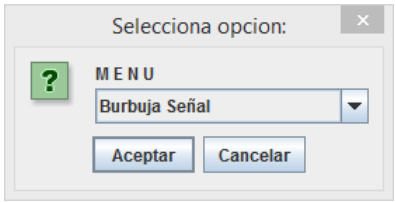
Complejidad espacial:

- La complejidad espacial del método de ordenamiento de doble burbuja con señal es O(1), lo que significa que no requiere espacio adicional en función del tamaño de la lista. El algoritmo opera directamente sobre la lista de entrada y realiza las comparaciones e intercambios en su lugar, sin utilizar estructuras de datos adicionales.

En resumen, la complejidad en el tiempo del método de ordenamiento de doble burbuja con señal es O(n^2) en el peor caso y en el caso medio, mientras que en el mejor caso tiene una complejidad de O(n). La complejidad espacial es constante, O(1), independientemente del tamaño de la lista.

Prueba de escritorio:





METODO DE ORDENAMIENTO SHELL

Análisis de eficiencia

Se considera la generalización del algoritmo de ordenación por burbujas o un algoritmo de ordenación por inserción optimizado. En el algoritmo de ordenación por inserción, movemos los elementos una posición hacia adelante.

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

Caso medio

La complejidad depende de la secuencia elegida para la ordenación. La complejidad temporal es del orden de [Big Theta]: O(nlog(n)2).

El peor caso

La complejidad temporal en el peor de los casos para la ordenación en cascarón es siempre menor que igual a O(n2). Más concretamente, según el teorema de Poonen, viene dada por Θ(nlogn)2/(log n)2) o Θ(nlog n)2/log n) o Θ(n(log n)) o algo intermedio. La complejidad temporal en el peor de los casos es [Big O]: menos que igual a O(n2).

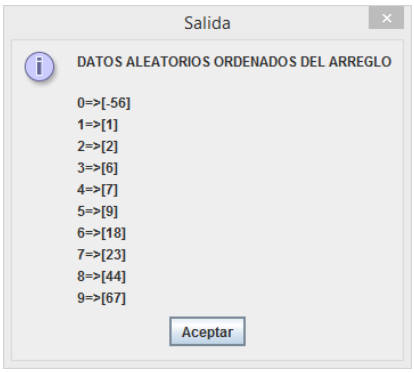
El mejor caso

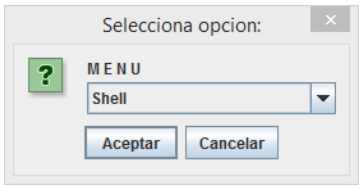
El mejor caso ocurre cuando el array ya está ordenado y las comparaciones necesarias para cada intervalo son iguales al tamaño del array. La complejidad temporal del mejor caso es [Big Omega]: O(nlogn).

Complejidad en el tiempo y complejidad espacial.

La complejidad en el tiempo es de O(n2/3)
La complejidad en el espacio es de O (n)

Prueba de escritorio:





METODO DE ORDENAMIENTO HEAPSHORT

Análisis de eficiencia

La ordenación en montón funciona de manera bastante similar a la ordenación por selección. Selecciona el elemento máximo de la matriz usando max-heap y lo coloca en su posición en la parte posterior de la matriz. Hace uso de un procedimiento llamado heapify()para construir el montón.

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

Caso promedio

La altura de un árbol binario completo con n elementos es máx logn. Entonces, la heapify()función puede tener un máximo de logn comparaciones cuando un elemento se mueve de la raíz a la hoja. La función de montón de compilación se llama para n/2elementos que hacen la complejidad de tiempo total para la primera etapa n/2*logno T(n)= nlogn.

HeapSort()toma lognel peor tiempo para cada elemento, y nlos elementos también hacen que su tiempo sea complejo nlogn. Tanto la complejidad de tiempo para construir el montón como la ordenación del montón se agregan y nos dan la complejidad resultante como nlogn. Por lo tanto, la complejidad temporal total es del orden de [Big Theta]: O(nlogn).

Peor de los casos

La complejidad temporal en el peor de los casos es [O grande]: O(nlogn).

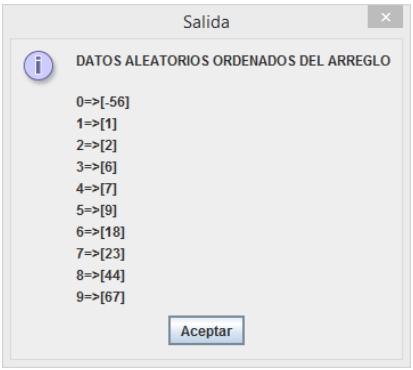
Mejor caso

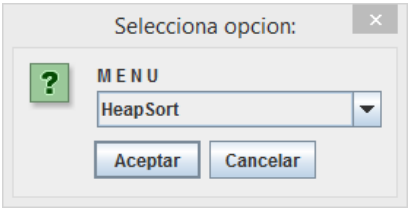
La complejidad de tiempo en el mejor de los casos es [Big Omega]: O(nlogn). Es lo mismo que la complejidad temporal del peor de los casos.

Complejidad en el tiempo y complejidad espacial.
La complejidad en el tiempo es de O(nlogn)

La complejidad en el espacio es de O(1)

Prueba de escritorio:





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.

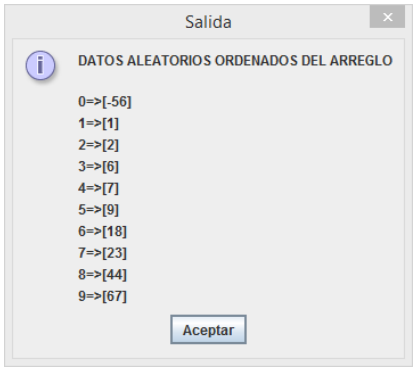
Tiempo:

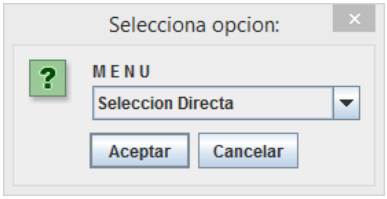
El análisis del algoritmo de selección es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño de la lista o array y no de la distribución inicial de los datos.

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.

Prueba de escritorio:





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

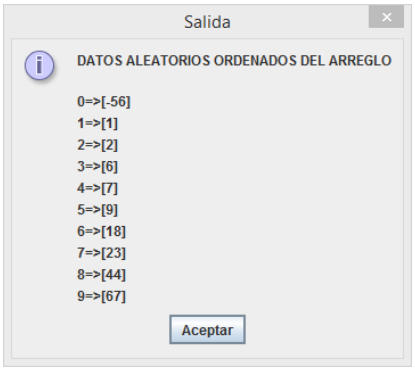
Complejidad en el tiempo y complejidad espacial.

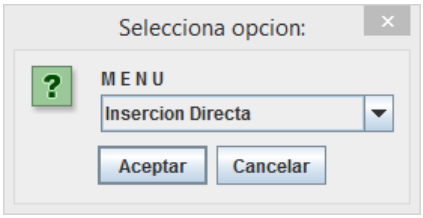
Tiempo: Caso de borde: cuando la posición del mínimo es k==i. Entonces se intercambia a[i] con a[i]. Una rápida revisión del intercambio permite apreciar que en ese caso a[i] no altera su valor, y por lo tanto no es incorrecto.

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.

Prueba de escritorio:





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).

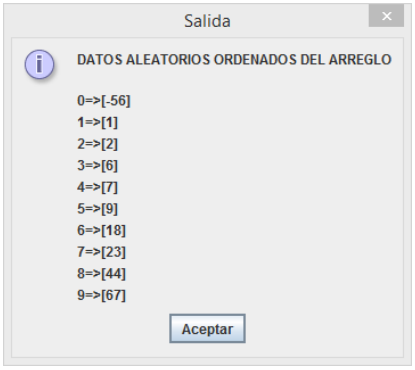
- El mejor caso

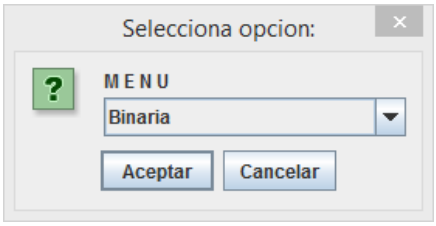
El mejor caso se produce cuando el array ya está ordenado y no es necesario desplazar elementos. La complejidad temporal en el mejor caso es [Big Omega]: O(n).

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.

Prueba de escritorio:





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·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·log n).

Complejidad en el tiempo y complejidad espacial.

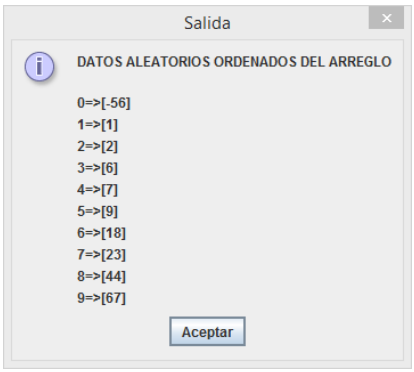
Complejidad en el tiempo:

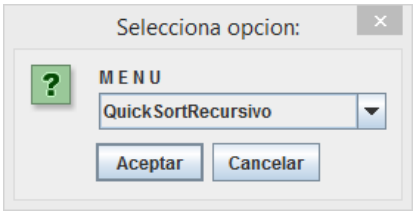
- En el caso promedio, Quicksort tiene una complejidad de tiempo promedio de O(n log n), donde "n" es el número de elementos a ordenar. Esto significa que el tiempo de ejecución del algoritmo crece de manera logarítmica con respecto al tamaño de la entrada. Quicksort logra esto dividiendo el conjunto de elementos en particiones más pequeñas y luego ordenándolas de forma recursiva. En promedio, Quicksort divide la lista aproximadamente por la mitad en cada iteración, lo que resulta en un tiempo de ejecución eficiente.
- En el peor caso, cuando la lista de entrada ya está ordenada o casi ordenada, y se elige el elemento pivot de manera desfavorable, Quicksort puede tener una complejidad de tiempo cuadrática de O(n^2). Esto ocurre cuando el pivot elegido siempre es el elemento mínimo o máximo, y el algoritmo no logra dividir la lista de manera equilibrada. Sin embargo, el peor caso se puede evitar mediante la elección adecuada del pivot, como seleccionar uno aleatoriamente o utilizar un enfoque de selección mediana.
- En el mejor caso, Quicksort tiene una complejidad de tiempo de O(n log n). Esto ocurre cuando el pivot elegido divide la lista en dos particiones aproximadamente iguales en cada iteración, lo que resulta en un rendimiento óptimo del algoritmo.

Complejidad espacial:

- La complejidad espacial de Quicksort es O(log n) en el peor caso debido a la recursión utilizada para dividir y ordenar las particiones. Esto se debe a que el algoritmo necesita mantener un stack de llamadas recursivas para realizar las divisiones.
- En el mejor caso y en el caso promedio, la complejidad espacial también es O(log n), ya que las llamadas recursivas se realizan en una pila, que tiene un tamaño proporcional al número de divisiones realizadas.

Prueba de escritorio:





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 y complejidad espacial.

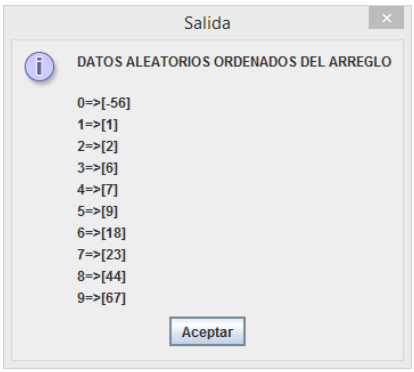
Complejidad en el tiempo:

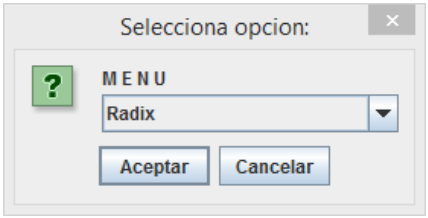
- La complejidad en el tiempo de Radix Sort depende del número de dígitos necesarios para representar los elementos a ordenar y del tamaño del conjunto de datos.
- En general, la complejidad en el tiempo de Radix Sort es O(d * (n + b)), donde "d" es el número de dígitos, "n" es el número de elementos 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.

Prueba de escritorio:





METODO DE INTERCALACION

Análisis de eficiencia

El algoritmo de intercalación (Merge Sort) tiene una eficiencia promedio y peor caso de O(n log n), donde "n" es el número de elementos que se están intercalando. Esto significa que el tiempo de ejecución aumenta de manera logarítmica con respecto al tamaño de la entrada. Veamos una explicación más detallada del análisis de eficiencia:

Complejidad temporal:

- Mejor caso: O(n log n) En el mejor caso, el algoritmo de intercalación realiza un número mínimo de comparaciones y movimientos de elementos, lo que se traduce en una eficiencia de O(n log n).
- Peor caso: O(n log n) En el peor caso, el algoritmo de intercalación realiza la máxima cantidad de comparaciones y movimientos de elementos. A pesar de esto, el tiempo de ejecución sigue siendo de O(n log n), lo cual es muy eficiente en comparación con otros algoritmos de ordenamiento.
- Caso promedio: O(n log n) El caso promedio también tiene una eficiencia de O(n log n), lo que indica que el algoritmo de intercalación es consistente en su rendimiento para diferentes conjuntos de datos.

Complejidad espacial:

- El algoritmo de intercalación requiere una cantidad adicional de espacio para almacenar los elementos intermedios mientras se realiza la intercalación. Por lo tanto, la complejidad espacial es de O(n), donde "n" es el número de elementos que se están intercalando.

En resumen, el algoritmo de intercalación (Merge Sort) en Java tiene una eficiencia promedio y peor caso de O(n log n). Esto lo convierte en una opción eficiente para ordenar grandes conjuntos de datos. Además, su tiempo de ejecución se mantiene

relativamente constante en diferentes situaciones, lo que lo hace predecible y confiable en términos de rendimiento.

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

Mejor caso: En el mejor caso, el arreglo ya está ordenado y no es necesario realizar ninguna comparación adicional. El algoritmo simplemente dividirá el arreglo en subarreglos y los fusionará sin necesidad de intercambiar elementos.

Ejemplo: Supongamos que tenemos el siguiente arreglo ordenado de manera ascendente: [1, 2, 3, 4, 5]. Al aplicar el algoritmo de intercalación, no se realizarán intercambios de elementos y el arreglo permanecerá igual.

Código en Java:

javaCopy code

```
int[] arr = {1, 2, 3, 4, 5};
mergeSort(arr);
```

En este caso, el tiempo de ejecución será de O(n log n), ya que se realizarán divisiones y fusiones, pero sin intercambios adicionales.

Caso medio: En el caso medio, el arreglo no está completamente ordenado ni completamente desordenado. El algoritmo realizará comparaciones y fusiones de los subarreglos para obtener el arreglo final ordenado.

Ejemplo: Consideremos el siguiente arreglo desordenado: [5, 2, 4, 1, 3]. Al aplicar el algoritmo de intercalación, se realizarán comparaciones y movimientos de elementos para ordenar el arreglo.

Código en Java:

javaCopy code

```
int[] arr = {5, 2, 4, 1, 3};
mergeSort(arr);
```

En este caso, el tiempo de ejecución también será de O(n log n), ya que se realizarán divisiones y fusiones con un número moderado de comparaciones y movimientos de elementos.

Peor caso: En el peor caso, el arreglo está completamente desordenado y se requieren múltiples comparaciones y movimientos de elementos para ordenarlo.

Ejemplo: Supongamos que tenemos el siguiente arreglo en orden descendente: [5, 4, 3, 2, 1]. Al aplicar el algoritmo de intercalación, se realizarán múltiples comparaciones y movimientos de elementos para ordenar el arreglo.

Código en Java:

javaCopy code

```
int[] arr = {5, 4, 3, 2, 1};
mergeSort(arr);
```

En este caso, el tiempo de ejecución seguirá siendo de O(n log n), pero el número de comparaciones y movimientos de elementos será mayor en comparación con los casos anteriores.

En resumen, el algoritmo de intercalación (Merge Sort) en Java tiene una eficiencia de O(n log n) en el mejor, caso medio y peor caso. La diferencia radica en la cantidad de comparaciones y movimientos de elementos necesarios en cada caso. En el mejor caso, el arreglo ya está ordenado y no se necesitan intercambios adicionales, mientras que en el peor caso, el arreglo está completamente desordenado y se requieren más comparaciones y movimientos para ordenarlo.

Complejidad en el tiempo y complejidad espacial

- Complejidad en el tiempo:
  - Mejor caso: O(n log n)
  - Caso medio: O(n log n)
  - Peor caso: O(n log n)

Esto significa que el tiempo de ejecución del algoritmo de intercalación es proporcional a "n log n", donde "n" es el número de elementos que se están intercalando. La eficiencia del algoritmo es muy buena y se mantiene constante en diferentes casos.

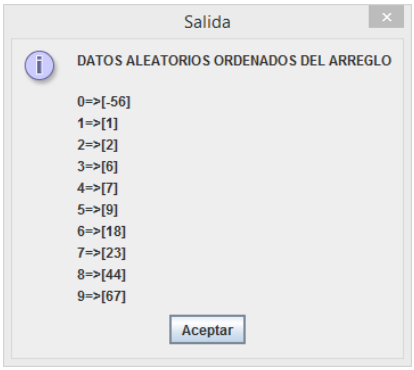
- Complejidad espacial: La complejidad espacial del algoritmo de intercalación es de O(n), donde "n" es el número de elementos que se están intercalando. Esto se debe a que se requiere espacio adicional para almacenar los subarreglos y los elementos intermedios durante el proceso de intercalación.

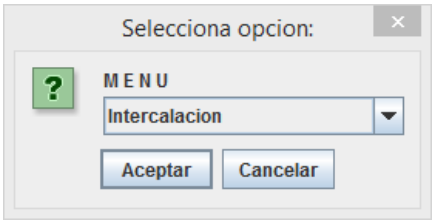
La complejidad espacial de O(n) implica que el algoritmo utiliza una cantidad de memoria proporcional al tamaño del arreglo de entrada. Esto es necesario para realizar las divisiones y fusiones de los subarreglos.

Es importante tener en cuenta que, aunque la complejidad espacial es de O(n), el algoritmo de intercalación de Merge Sort es eficiente en términos de uso de memoria en comparación con otros algoritmos de ordenamiento que requieren estructuras de datos auxiliares más grandes.

En resumen, la complejidad en el tiempo del algoritmo de intercalación en Java es de O(n log n) en el mejor, caso medio y peor caso, mientras que la complejidad espacial es de O(n). Estas complejidades aseguran un rendimiento eficiente en la mayoría de los casos y una utilización razonable de memoria.

Prueba de escritorio:





METODO DE MEZCLA DIRECTA

Análisis de eficiencia

La eficiencia de la mezcla directa (también conocida como "Insertion Sort") en Java depende del tamaño del arreglo a ordenar. Aquí tienes el análisis de eficiencia de este algoritmo:

Complejidad temporal:

- Mejor caso: O(n) En el mejor caso, cuando el arreglo ya está ordenado o casi ordenado, la mezcla directa realiza una cantidad mínima de comparaciones y movimientos de elementos. En este caso, el algoritmo tiene una eficiencia lineal de O(n).
- Peor caso: O(n^2) En el peor caso, cuando el arreglo está ordenado en orden inverso, la mezcla directa realiza una cantidad máxima de comparaciones y movimientos de elementos. En este caso, el algoritmo tiene una eficiencia cuadrática de O(n^2).
- Caso promedio: O(n^2) En general, la mezcla directa tiene una eficiencia cuadrática en el caso promedio. Esto significa que el tiempo de ejecución aumenta cuadráticamente con el tamaño del arreglo.

Complejidad espacial: La complejidad espacial de la mezcla directa es de O(1) en el peor caso. El algoritmo no requiere espacio adicional para almacenar subarreglos o estructuras de datos auxiliares. Los movimientos de elementos se realizan directamente en el arreglo de entrada.

En resumen, la mezcla directa en Java tiene una eficiencia de O(n) en el mejor caso, O(n^2) en el peor caso y una eficiencia cuadrática en el caso promedio. A diferencia

de otros algoritmos de ordenamiento más eficientes, como el algoritmo de intercalación (Merge Sort) o el algoritmo de ordenamiento rápido (QuickSort), la mezcla directa es menos eficiente en términos de tiempo de ejecución. Sin embargo, tiene la ventaja de no requerir memoria adicional y puede ser útil para ordenar arreglos pequeños o casi ordenados.

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

Mejor caso: En el mejor caso, el arreglo ya está ordenado o casi ordenado, lo que minimiza el número de comparaciones y movimientos de elementos.

Ejemplo: Supongamos que tenemos el siguiente arreglo ordenado de manera ascendente: [1, 2, 3, 4, 5]. Al aplicar la mezcla directa, no se realizarán muchos movimientos de elementos, ya que el arreglo ya está ordenado.

Código en Java:

javaCopy code

```
int[] arr = {1, 2, 3, 4, 5};
insertionSort(arr);
```

En este caso, el tiempo de ejecución será de O(n), ya que se realizarán comparaciones mínimas y se evitarán movimientos de elementos.

Caso medio: En el caso medio, el arreglo no está completamente ordenado ni completamente desordenado. La mezcla directa realizará comparaciones y movimientos de elementos para ordenar el arreglo.

Ejemplo: Consideremos el siguiente arreglo desordenado: [5, 2, 4, 1, 3]. Al aplicar la mezcla directa, se realizarán comparaciones y movimientos de elementos para ordenar el arreglo.

Código en Java:

javaCopy code

```
int[] arr = {5, 2, 4, 1, 3};
insertionSort(arr);
```

En este caso, el tiempo de ejecución será de O(n^2), ya que se requerirán comparaciones y movimientos de elementos para ordenar el arreglo.

Peor caso: En el peor caso, el arreglo está completamente desordenado y se requieren la máxima cantidad de comparaciones y movimientos de elementos para ordenarlo.

Ejemplo: Supongamos que tenemos el siguiente arreglo en orden descendente: [5, 4, 3, 2, 1]. Al aplicar la mezcla directa, se realizarán múltiples comparaciones y movimientos de elementos para ordenar el arreglo.

Código en Java:

javaCopy code

```
int[] arr = {5, 4, 3, 2, 1};
insertionSort(arr);
```

En este caso, el tiempo de ejecución será de O(n^2), ya que se realizarán muchas comparaciones y movimientos para ordenar el arreglo.

En resumen, la mezcla directa (Insertion Sort) en Java tiene una eficiencia de O(n) en el mejor caso, O(n^2) en el caso medio y peor caso. Es menos eficiente que otros algoritmos de ordenamiento como el algoritmo de intercalación (Merge Sort) o el algoritmo de ordenamiento rápido (QuickSort). Sin embargo, puede ser útil para arreglos pequeños o casi ordenados debido a su implementación simple y la falta de necesidad de memoria adicional.

Complejidad en el tiempo y complejidad espacial

La complejidad en el tiempo y la complejidad espacial del algoritmo de mezcla directa (Insertion Sort) en Java son las siguientes:

Complejidad en el tiempo:

- Mejor caso: O(n) En el mejor caso, cuando el arreglo ya está ordenado o casi ordenado, la mezcla directa realiza una cantidad mínima de comparaciones y movimientos de elementos. En este caso, el algoritmo tiene una eficiencia lineal de O(n).
- Peor caso: O(n^2) En el peor caso, cuando el arreglo está ordenado en orden inverso, la mezcla directa realiza una cantidad máxima de comparaciones y movimientos de elementos. En este caso, el algoritmo tiene una eficiencia cuadrática de O(n^2).
- Caso promedio: O(n^2) En general, la mezcla directa tiene una eficiencia cuadrática en el caso promedio. Esto significa que el tiempo de ejecución aumenta cuadráticamente con el tamaño del arreglo.

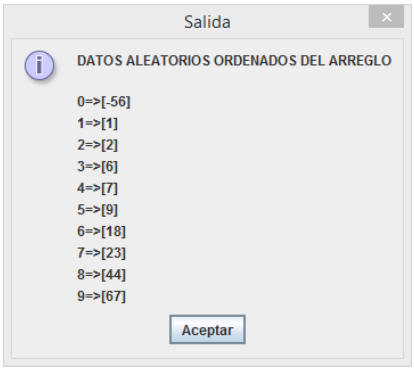
Complejidad espacial:

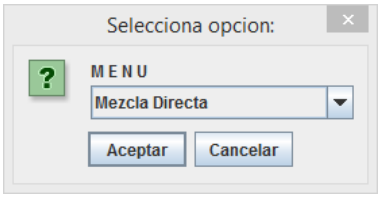
La complejidad espacial de la mezcla directa es de O(1) en el peor caso. El algoritmo realiza los movimientos de elementos directamente en el arreglo de entrada sin necesidad de memoria adicional. Por lo tanto, no requiere espacio adicional en función del tamaño del arreglo.

Es importante tener en cuenta que la mezcla directa tiene una complejidad en el tiempo menos eficiente que otros algoritmos de ordenamiento como el algoritmo de intercalación (Merge Sort) o el algoritmo de ordenamiento rápido (QuickSort). Sin embargo, la complejidad espacial es muy eficiente, ya que no se requiere memoria adicional más allá del arreglo original.

En resumen, la complejidad en el tiempo del algoritmo de mezcla directa en Java es de O(n) en el mejor caso, O(n^2) en el peor caso y una eficiencia cuadrática en el caso promedio. La complejidad espacial es de O(1), lo que implica un uso eficiente de la memoria.

Prueba de escritorio:





METODO DE MEZCLA NATURAL

Análisis de eficiencia

El algoritmo de mezcla natural (Natural Merge Sort) es una variante del algoritmo de mezcla (Merge Sort) que aprovecha secuencias ordenadas existentes en el arreglo para mejorar la eficiencia. A continuación, se presenta el análisis de eficiencia de la mezcla natural en Java:

Complejidad en el tiempo:

- Mejor caso: O(n) En el mejor caso, cuando el arreglo ya está ordenado, la mezcla natural realiza una pasada para detectar las secuencias ordenadas y no necesita realizar comparaciones adicionales. En este caso, el algoritmo tiene una eficiencia lineal de O(n).
- Peor caso: O(n^2) En el peor caso, cuando el arreglo está ordenado en orden inverso, la mezcla natural necesita realizar múltiples pasadas para fusionar las secuencias ordenadas. En este caso, el algoritmo tiene una eficiencia cuadrática de O(n^2).
- Caso promedio: O(n log n) En general, la mezcla natural tiene una eficiencia de O(n log n) en el caso promedio. Esto se debe a que, en

promedio, se necesitarán múltiples pasadas para fusionar todas las secuencias ordenadas y lograr el arreglo completamente ordenado.

Complejidad espacial: La complejidad espacial de la mezcla natural es de O(n), ya que se requiere espacio adicional para almacenar los subarreglos y los elementos intermedios durante el proceso de mezcla. Esto se debe a que el algoritmo necesita crear subarreglos temporales para fusionar las secuencias ordenadas.

En resumen, la mezcla natural en Java tiene una eficiencia de O(n) en el mejor caso, O(n^2) en el peor caso y una eficiencia de O(n log n) en el caso promedio. La complejidad espacial es de O(n), lo que implica un uso adicional de memoria proporcional al tamaño del arreglo original. La mezcla natural puede ser una opción eficiente cuando se trabaja con arreglos parcialmente ordenados, ya que aprovecha las secuencias ordenadas existentes. Sin embargo, en el peor caso, su eficiencia es comparable a otros algoritmos de ordenamiento cuadráticos.

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

Mejor caso: En el mejor caso, el arreglo ya está completamente ordenado y el algoritmo de mezcla natural aprovecha las secuencias ordenadas para realizar menos comparaciones y movimientos de elementos.

Ejemplo: Supongamos que tenemos el siguiente arreglo ordenado de manera ascendente: [1, 2, 3, 4, 5]. Al aplicar la mezcla natural, se detectará que el arreglo está ordenado y no se necesitará realizar ninguna comparación adicional.

Código en Java:

javaCopy code

```
int[] arr = {1, 2, 3, 4, 5};
naturalMergeSort(arr);
```

En este caso, el tiempo de ejecución será de O(n), ya que el algoritmo detecta rápidamente que el arreglo ya está ordenado.

Caso medio: En el caso medio, el arreglo no está completamente ordenado ni completamente desordenado. El algoritmo de mezcla natural realizará múltiples pasadas para fusionar las secuencias ordenadas y lograr el arreglo ordenado.

Ejemplo: Consideremos el siguiente arreglo desordenado: [5, 2, 4, 1, 3]. Al aplicar la mezcla natural, se realizarán varias pasadas para fusionar las secuencias ordenadas hasta obtener el arreglo ordenado.

Código en Java:

javaCopy code

```
int[] arr = {5, 2, 4, 1, 3};
naturalMergeSort(arr);
```

En este caso, el tiempo de ejecución será de O(n log n), ya que se necesitarán múltiples pasadas para fusionar todas las secuencias ordenadas y obtener el arreglo ordenado.

Peor caso: En el peor caso, el arreglo está ordenado en orden inverso y se necesitarán múltiples pasadas para fusionar las secuencias ordenadas y lograr el arreglo ordenado.

Ejemplo: Supongamos que tenemos el siguiente arreglo en orden descendente: [5, 4, 3, 2, 1]. Al aplicar la mezcla natural, se realizarán múltiples pasadas para fusionar las secuencias ordenadas y obtener el arreglo ordenado.

Código en Java:

javaCopy code

```
int[] arr = {5, 4, 3, 2, 1};
naturalMergeSort(arr);
```

En este caso, el tiempo de ejecución será de O(n^2), ya que se necesitarán muchas pasadas para fusionar todas las secuencias ordenadas y obtener el arreglo ordenado.

En resumen, la mezcla natural (Natural Merge Sort) en Java tiene una eficiencia de O(n) en el mejor caso, O(n log n) en el caso medio y una eficiencia de O(n^2) en el peor caso. El algoritmo aprovecha las secuencias ordenadas para mejorar la eficiencia, pero en el peor caso, su rendimiento es comparable a otros algoritmos de ordenamiento cuadráticos.

Complejidad en el tiempo y complejidad espacial

Complejidad en el tiempo:

- Mejor caso: O(n) En el mejor caso, cuando el arreglo ya está completamente ordenado, la mezcla natural detecta rápidamente las secuencias ordenadas y no necesita realizar ninguna comparación adicional. En este caso, el algoritmo tiene una eficiencia lineal de O(n).
- Peor caso: O(n^2) En el peor caso, cuando el arreglo está ordenado en orden inverso, la mezcla natural necesita realizar múltiples pasadas

para fusionar las secuencias ordenadas y lograr el arreglo ordenado. En este caso, el algoritmo tiene una eficiencia cuadrática de O(n^2).

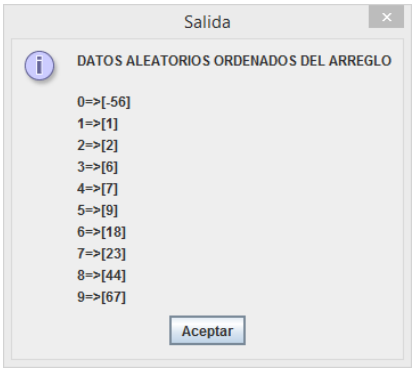
- Caso promedio: O(n log n) En general, la mezcla natural tiene una eficiencia de O(n log n) en el caso promedio. Esto se debe a que, en promedio, se necesitarán múltiples pasadas para fusionar todas las secuencias ordenadas y lograr el arreglo ordenado.

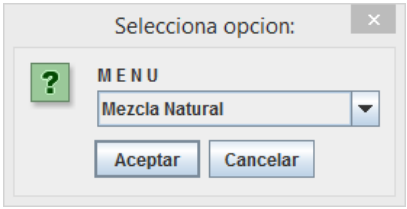
Complejidad espacial:

La complejidad espacial de la mezcla natural es de O(n), ya que se requiere espacio adicional para almacenar los subarreglos temporales durante el proceso de mezcla. A medida que se fusionan las secuencias ordenadas, se crean subarreglos temporales que contienen los elementos intermedios. Por lo tanto, se necesita espacio adicional proporcional al tamaño del arreglo original.

En resumen, la complejidad en el tiempo del algoritmo de mezcla natural en Java es de O(n) en el mejor caso, O(n^2) en el peor caso y una eficiencia de O(n log n) en el caso promedio. La complejidad espacial es de O(n), ya que se requiere espacio adicional para los subarreglos temporales. La mezcla natural es una variante del algoritmo de mezcla (Merge Sort) que aprovecha las secuencias ordenadas para mejorar la eficiencia en comparación con el Merge Sort tradicional. Sin embargo, su rendimiento sigue siendo afectado por el orden inicial del arreglo.

Prueba de escritorio:





IMPLEMENTACION DE METODOS

METODO ARRAY VACIO

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

Este método nos regresara un verdadero o falso si el arreglo esta vacío, 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

```
@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

```
@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

```
@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

```
@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

```
@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);
}
```

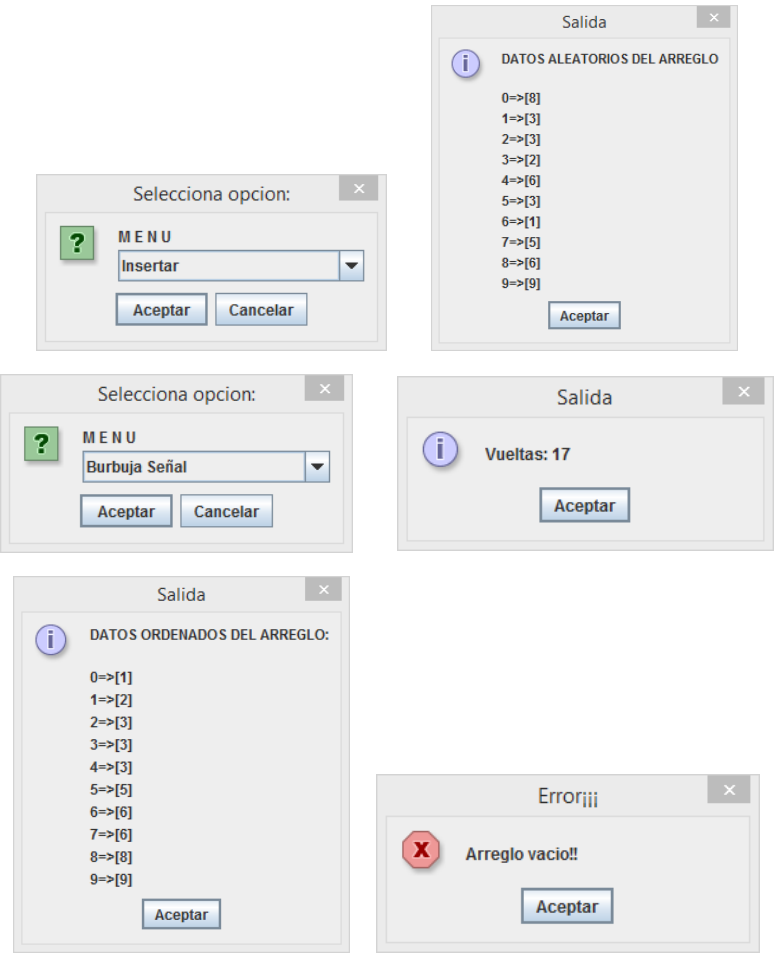
En este método llamado burbuja con señal lo que vamos a hacer es tener una variable booleana para que nos permita hacer funcionar el código, usaremos 2 arreglos uno que incremente en i, y otro que incremente en j, por lo cual en j hacemos todo el ordenamiento de los datos, ahí es cuando entra la variable band que ahora se encuentra con verdadero, que quiere decir que si es correcto lo que hace, después incrementamos las vueltas que dio el ordenamiento, y estas mismas se mostraran en pantalla. Después en otro if vamos a negar band, y si es así no hará nada solo dar un break y termina.

METODO BURBUJA CON SEÑAL EN MENU

```
case "Burbuja Señal":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {
    ordena.burbujaSeñal();
    Tools.imprime("DATOS ORDENADOS DEL ARREGLO:\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método de burbuja con señal, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
}
```

En este método llamado burbuja con señal lo que hace es mejorar el tiempo comparando cada elemento con el que está a un cierto número de posiciones

llamado salto, en lugar de compararlo con el que está justo a su lado. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera).

Se van dando pasadas con el mismo salto hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo:

Tenemos un array {50,26,7,9,15,27}

Hacemos un salto de 3, entonces en este caso el 9 y el 50 se intercambian. Después el 15 y el 26 se intercambian. Posteriormente hace un salto de 1, y en este se intercambian el 15 y el 7, después el 50 y 26 se intercambian, e igual se intercambian el 50 y 27.

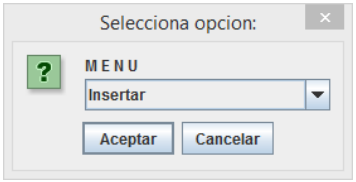
En la segunda pasada con salto 1, se intercambian el 9 y 7, y entonces finalmente el array esta ordenado. Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

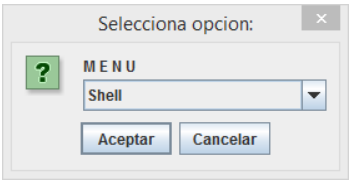
METODO SHELL INCREMENTOS-DECREMENTOS EN MENU

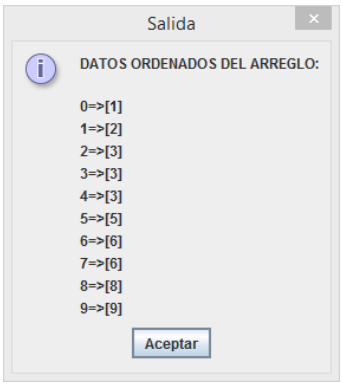
```
case "Shell":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {ordena.shellIncreDecre();
    Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());}
break;
```

Ya que terminamos el método de Shell, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
}
```

En este método llamado selección directa lo que hace es buscar el elemento mas pequeño del arreglo y así se coloca en la primera posición, entre las restantes se busca el elemento mas pequeño también, y se coloca en la segunda posición, y así sucesivamente hasta que se llegue a colocar el ultimo elemento. Esto es una mejor forma clara de explicarlo.

Por ejemplo:

Tenemos un array {50,26,7,9,15,27}

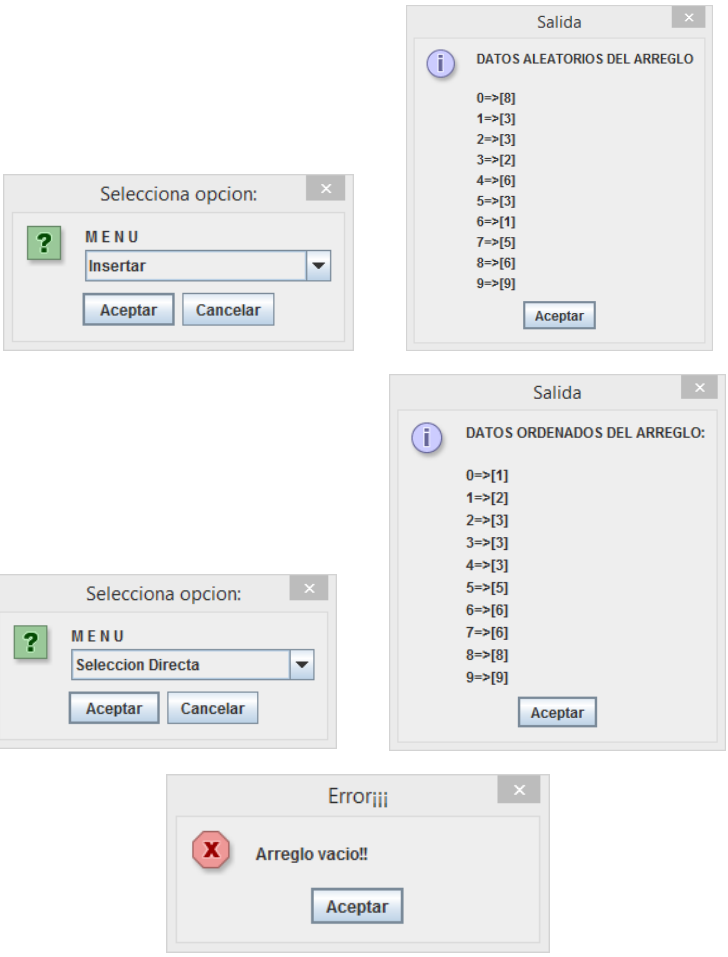
Se coloca el 7 en la primera posición, se intercambian el 7 y el 50. Después el 9 se coloca en la segunda posición e igual se intercambia el 9 pero ahora con el 25. Posteriormente el 15 se coloca en la tercera posición, para después este mismo intercambiarse con el 50, luego el menor de los que quedan es el 26, por lo cual se queda en su posición. Finalmente el 27 se coloca en la quinta posición, para luego este mismo intercambiarse con el 50, y así finalmente queda ordenado el arreglo.

METODO SELECCIÓN DIRECTA EN MENU

```
case "Seleccion Directa":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {ordena.seleDirecta();
    Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());}
break;
```

Ya que terminamos el método de selección directa, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
}
```

En este método llamado inserción directa lo que hace es 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.

Esto es una mejor forma clara de explicarlo.

Por ejemplo:

Tenemos un array {30,15,2,21,44,8}

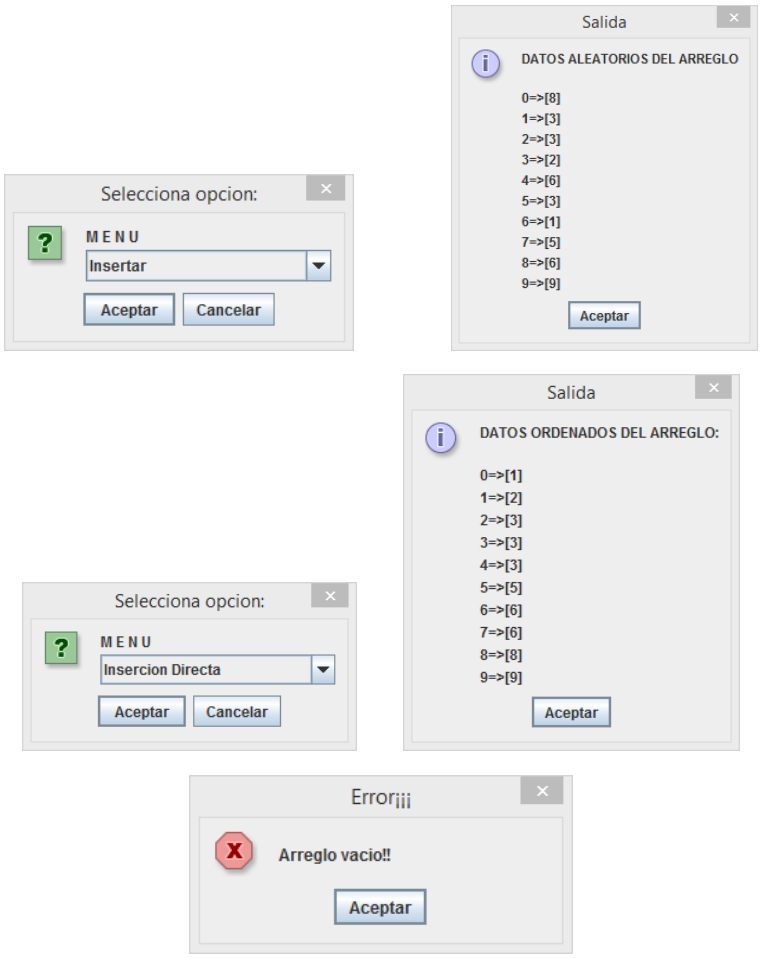
Se empieza por el segundo elemento, y se compara con el primero. Como 15 es menor a 30 se desplaza el 30 hacia la derecha y el 15 se coloca en su lugar. Seguimos por el tercer elemento, se compara con los anteriores y se van desplazando hasta que el 2 queda en su lugar. Continuamos por el cuarto elemento, este se compara con los anteriores y se van desplazando hasta que el 21 queda en su lugar. Lo mismo para el quinto elemento, en este caso ya está en su posición correcta respecto a los anteriores y así finalmente se coloca el último elemento y el array queda ordenado el arreglo.

METODO INSERCIÓN DIRECTA EN MENU

```
case "Insercion Directa":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {ordena.inserDirecta();
    Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());}
break;
```

Ya que terminamos el método de inserción directa, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
}
```

En este método llamado binaria lo que hace es que dividimos la matriz en dos subarreglos: ordenados y no ordenados. El primer elemento del arreglo está en el subarreglo ordenado, y el resto de los elementos están en el no ordenado.

Luego iteramos desde el segundo elemento hasta el último elemento. Para la i-ésima iteración, hacemos que el elemento actual sea nuestro "clave". Esta clave es el elemento que tenemos que agregar a nuestro subarreglo ordenado existente.

Para hacer esto, primero usamos la búsqueda binaria en el subarreglo ordenado para encontrar la posición del elemento que es justo mayor que nuestra clave. Llamemos a esta posición "temp". Luego cambiamos a la derecha todos los elementos desde la posición pos a i-1 y luego hacemos datos[pos+1] = temp..

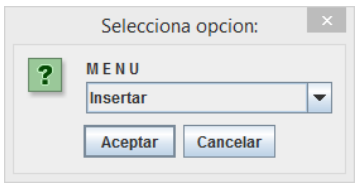
Podemos notar que para cada i-ésima iteración, la parte izquierda de la matriz hasta (i-1) siempre está ordenada. Y posteriormente imprimimos los datos ordenados de dicho método.

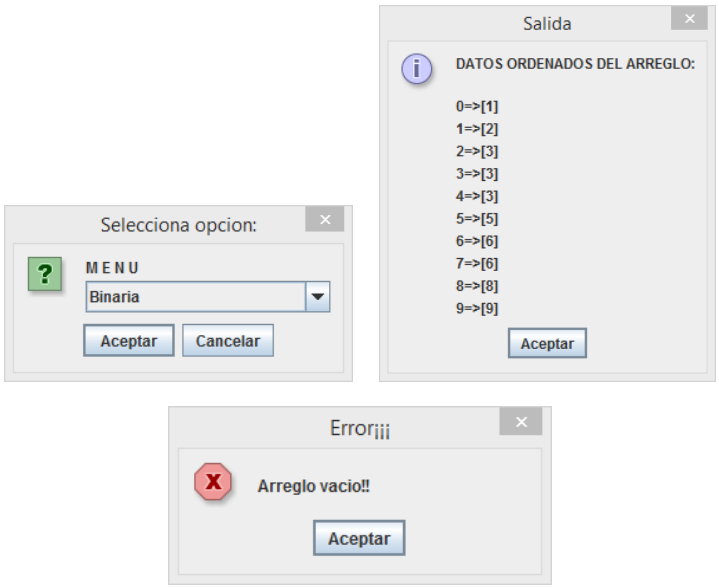
METODO BINARIA EN MENU

```
case "Binaria":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {ordena.binaria();
    Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());  }
break;
```

Ya que terminamos el método binaria, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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);}
}
```

En este método llamado HeapSort lo que hace es que encontramos el elemento mínimo y colocamos el elemento mínimo al principio. Repita el mismo proceso para los elementos restantes.

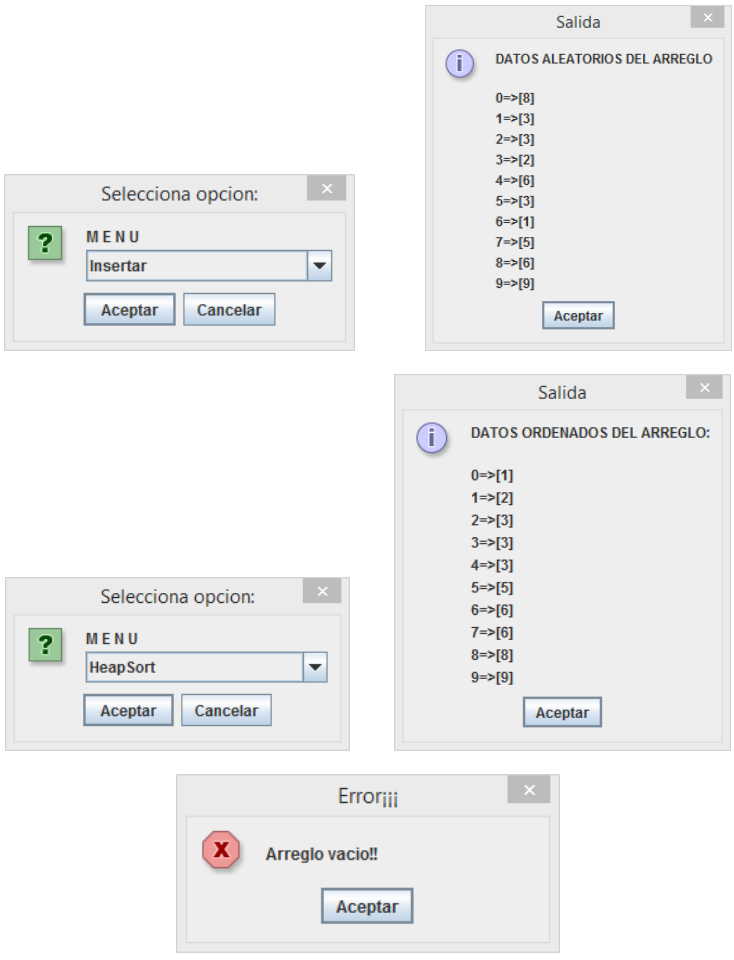
HeapSort2 es el proceso de crear una estructura de datos de montón a partir de un árbol binario representado mediante una matriz. Se utiliza para crear Min-Ha o Max-heap. Comience desde el último índice del nodo que no es hoja cuyo índice está dado por n/2 – 1. HeapSort2 usa la recursividad.

METODO HEAPSORT EN MENU

```
case "HeapSort":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {ordena.heapSort();
    Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método HeapSort, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
}

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;
}
```

En este método llamado radix lo que hace es que encuentre el elemento más grande en la matriz, es decir, máximo. Sea X el número de dígitos en max. X se calcula porque tenemos que pasar por todos los lugares significativos de todos los elementos. Este clasifica los elementos agrupando primero los dígitos individuales del mismo valor posicional . Luego, ordena los elementos según su orden creciente/decreciente.

Supongamos que tenemos una matriz de 8 elementos. Primero, ordenaremos los elementos según el valor de la unidad de lugar. Luego, ordenaremos los elementos según el valor del décimo lugar. Este proceso continúa hasta el último lugar significativo.

En esta matriz [121, 432, 564, 23, 1, 45, 788], tenemos el número más grande 788. Tiene 3 dígitos. Por lo tanto, el ciclo debe subir hasta el lugar de las centenas (3 veces).

Ahora, revisamos cada lugar significativo uno por uno. Utilizamos cualquier técnica de clasificación estable para clasificar los dígitos en cada lugar significativo. Hemos usado la ordenación por conteo para esto. Después ordenamos los elementos según los dígitos del lugar de la unidad (X=0).

Radix trabajando con Counting Sort como paso intermedio

Uso de clasificación por conteo para clasificar elementos según el lugar de la unidad Ahora, ordenamos los elementos según los dígitos en el lugar de las decenas. Ordenamos los elementos según el lugar de las decenas

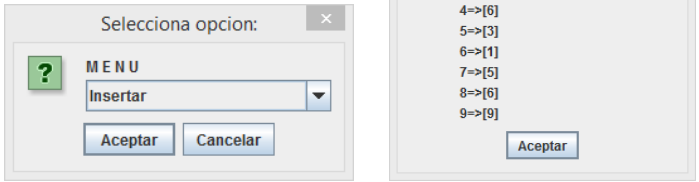
Finalmente, ordene los elementos según los dígitos en el lugar de las centenas..

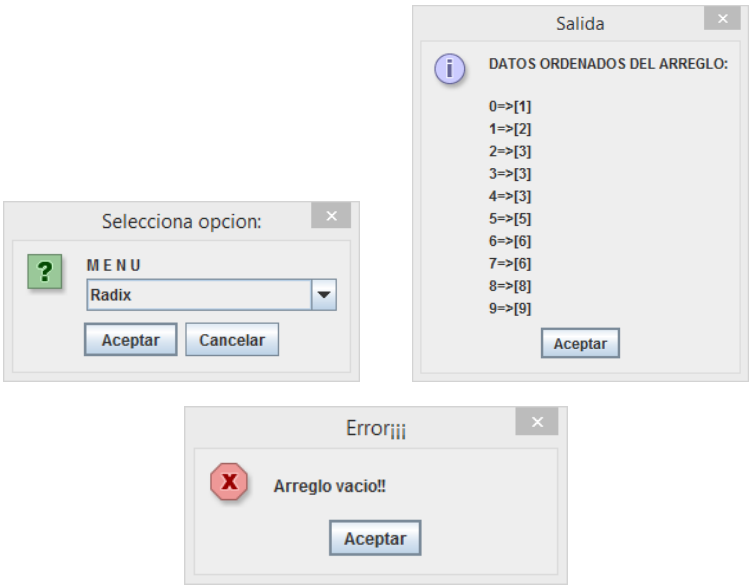
METODO RADIX EN MENU

```
case "Radix":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {
        ordena.radix();
        Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método radix, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
}
```

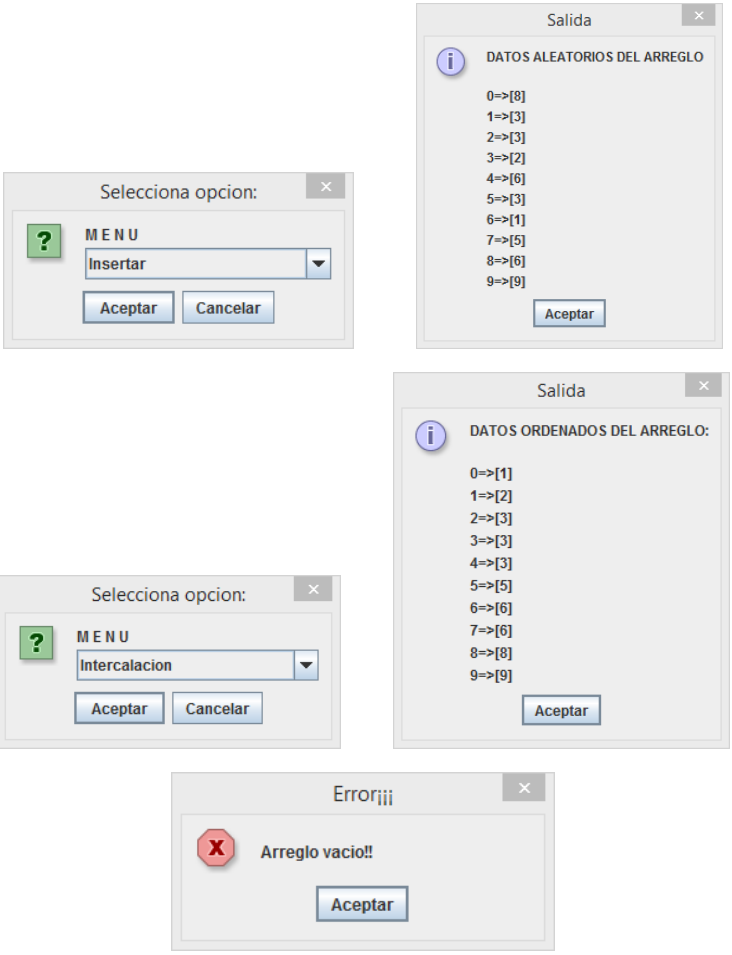
En este método llamado intercalación su objetivo es que por medio de una variable que almacenara los datos del arreglo, si n llegara a ser menor a 2 entonces no regresara nada, solo un return. En caso de que no entonces ira incrementando i y lo va a ir almacenando en datos, por otro lado también hará lo mismo en j, y finalmente lo que hay en datos se los asignara a elemento.

METODO INTERCALACIÓN EN MENU

```
case "Intercalacion":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {
        ordena.intercalacion();
        Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método intercalación, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacío, si lo está nos dirá que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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;
    }
```

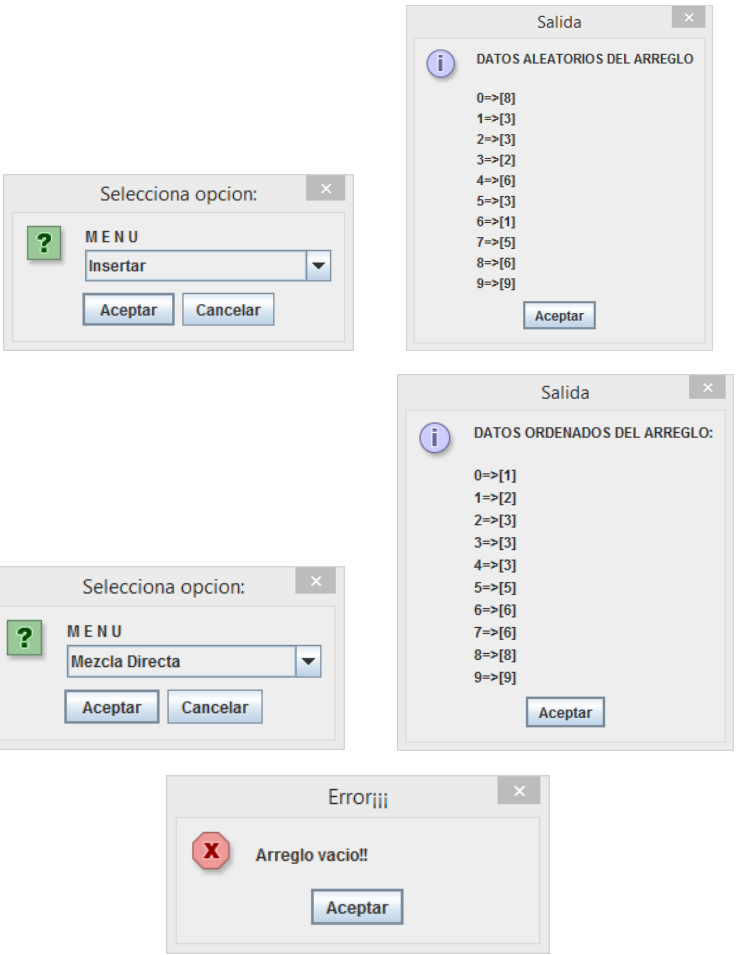
La idea central de este algoritmo consiste en la realización sucesiva de una partición y una fusión que produce secuencias ordenadas de longitud cada vez mayor. En la primera pasada la partición es de longitud 1, y la fusión o mezcla produce secuencias ordenadas de longitud 2. En la segunda pasada la partición es de longitud 2, y la fusión o mezcla produce secuencias ordenadas de longitud 4. Este proceso se repite hasta que la longitud de la secuencia de la secuencia para la partición sea mayor o igual que el número de elementos del archivo original. Todo lo anterior dicho, se va a poner un segundo método, el cual será recursivo, de tal manera que cuando nos encontremos en el método original llamaremos de forma indirecta el método que usamos para que así funcione nuestro método.

METODO MEZCLA DIRECTA EN MENU

```
case "Mezcla Directa":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {
        ordena.mezclaDirecta();
        Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método mezcla directa, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacio, si lo está nos dira que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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

```java
@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);

}
```

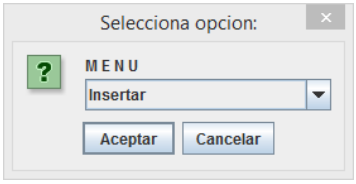

En este método en lo que consiste en aprovechar la existencia de secuencias ya ordenadas dentro de los datos del arreglo. A partir de las secuencias ordenadas existentes en el arreglo, se obtienen particiones que se almacenan en dos arreglos. Los datos almacenados en estos arreglos auxiliares se fusionan posteriormente para crear secuencias ordenadas cuya longitud se incrementa arbitrariamente hasta conseguir la total ordenación de los datos contenidos en el arreglo original.

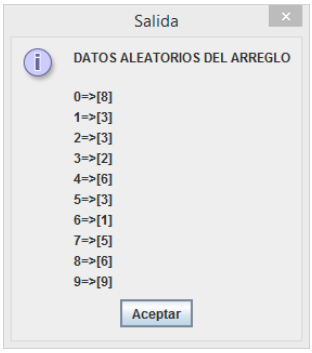
METODO MEZCLA NATURAL EN MENU

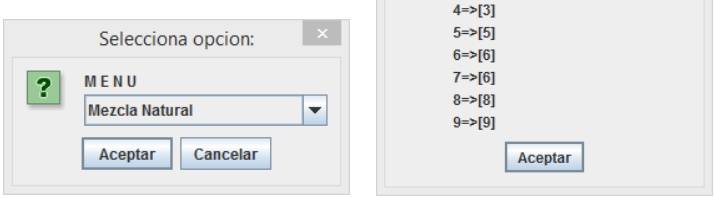
```java
case "Mezcla Natural":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {
        ordena.mezclaNatural();
        Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método mezcla natural, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacio, si lo está nos dira que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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);
    }
```

En este método lo que se hara es ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Después de elegir el pivote se realizan dos búsquedas:

- Una de izquierda a derecha, buscando un elemento mayor que el pivote
- Otra de derecha a izquierda, buscando un elemento menor que el pivote.
- Cuando se han encontrado los dos elementos anteriores, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.

La implementación del método de ordenación Quicksort es claramente recursiva.

Suponiendo que tomamos como pivote el primer elemento, el método Java Quicksort que implementa este algoritmo de ordenación para ordenar un array de enteros se presenta a continuación. Los parámetros izq y der son el primer y último elemento del array a tratar en cada momento.

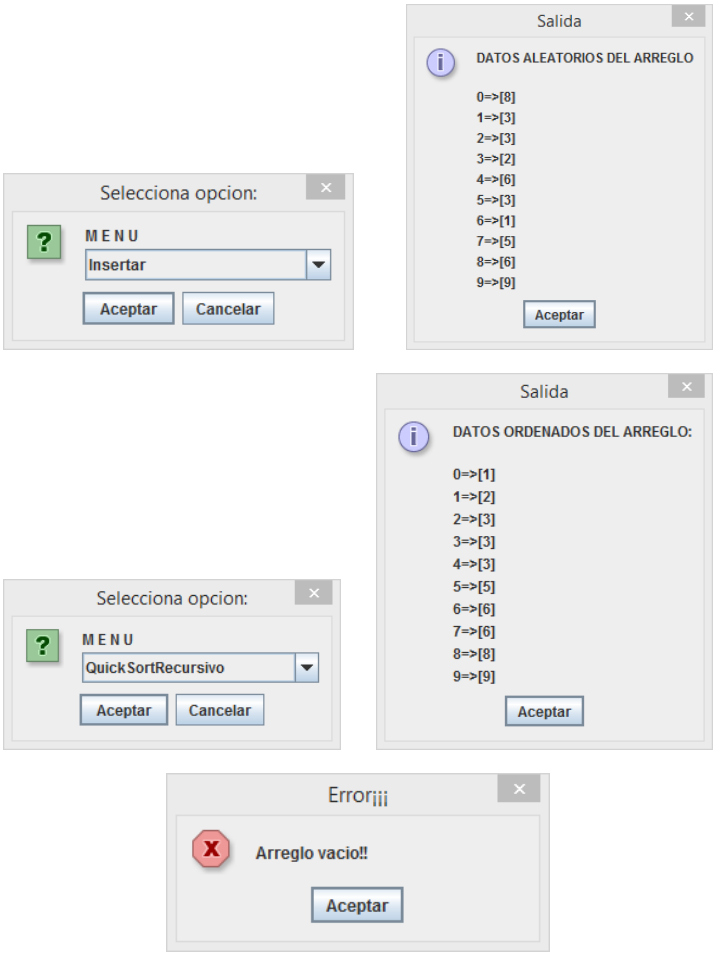
La elección del pivote determinará la eficiencia de este algoritmo ya que determina la partición del array. Si consideramos que el array está desordenado, podemos elegir el primer elemento y el algoritmo funcionaría de forma eficiente. Pero si el array está casi ordenado, elegir el primer elemento como pivote sería una mala solución ya que obtendríamos un subarray muy pequeño y otro muy grande. Por la misma razón, elegir el último elemento del array como pivote también es una mala idea. Pretendemos conseguir que el tamaño de los subarrays sea lo más parecido posible.

METODO QUICK RECURSIVO EN MENU

```
case "QuickSortRecursivo":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {ordena.quicksortRecursivo();
        Tools.imprime("DATOS ALEATORIOS ORDENADOS DEL ARREGLO\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método quickrecursivo, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacio, si lo está nos dira que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

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 DOBLE BURBUJA

```java
@Override
public void dobleBurbuja() {
    boolean band;
    for (int i = 0; i < datos.length / 2; i++) {
        band = false;
        for (int j = i; 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;
            }
        }
        if (!band) {
            break;
        }

        band = false;
        for (int j = datos.length - i - 2; j > i; j--) {
            if (datos[j] < datos[j - 1]) {
                int temp = datos[j];
                datos[j] = datos[j - 1];
                datos[j - 1] = temp;
                band = true;
            }
        }
        if (!band) {
            break;
        }
    }
}
```

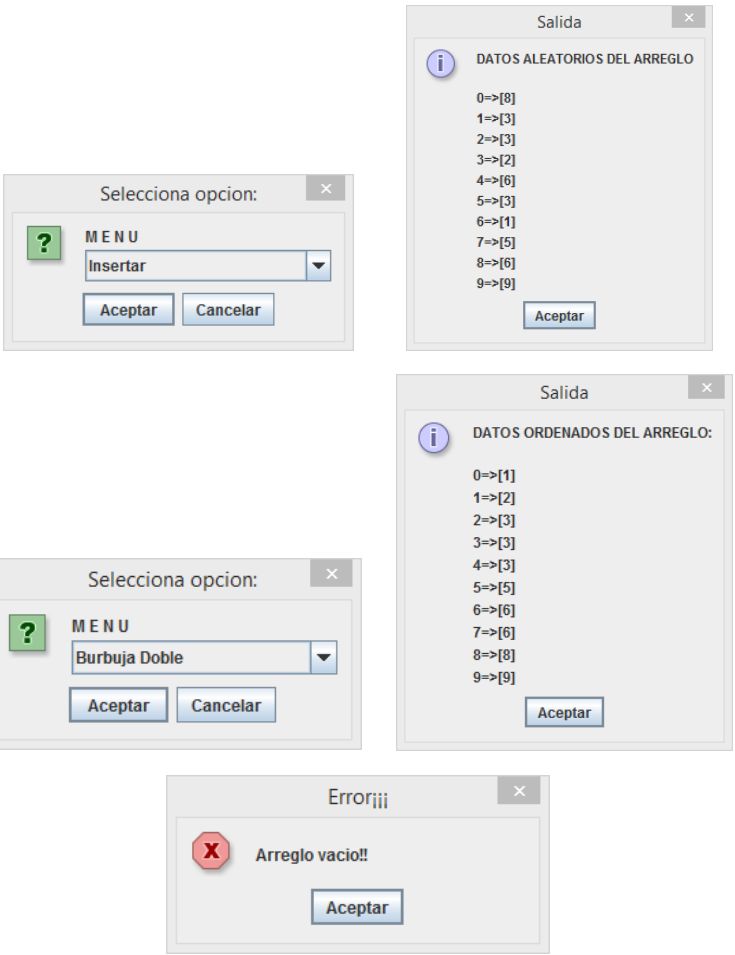
En este metodo lo que se hara es intentar mejorar el rendimiento del ordenamiento burbuja realizando el recorrido de comparación en ambas direcciones, de esta manera se puede realizar más de un intercambio por iteración. Y esto por medio de los dos arreglos vamos a ir comparando dato por dato para poderlo intercambiar de posicion, hasta el punto que la variable band sera verdadera. Asi sucesivamente hasta que el arreglo queda ordenado.

METODO DOBLE BURBUJA EN MENU

```java
case "Burbuja Doble":
    if(ordena.arrayVacio())Tools.imprimeErrorMsje("Arreglo vacio!!");
    else {
    ordena.dobleBurbuja();
    Tools.imprime("DATOS ORDENADOS DEL ARREGLO:\n"+ordena.imprimeDatos());
    }
break;
```

Ya que terminamos el método doble burbuja, entonces en la prueba de menú lo llamamos, para esto primero preguntamos si el array esta vacio, si lo está nos dira que no hay ningún dato, en caso contrario llamamos al método, y posteriormente imprimimos los datos ordenados de dicho método.

EJECUTABLES METODO DOBLE BURBUJA



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 SALIR

```
case "Salir":
    Tools.imprime("Fin del programa");
break;
```

En este método solo lo que hará es mandar un mensaje de que el programa finalizo.

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