TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO NACIONAL DE MEXICO CAMPUS ORIZABA

Materia:

Estructura De Datos

Maestra

Martínez Castillo María Jacinta

Integrantes:

Bravo Rosas Yamileth
Crescencio Rico José Armando
Gómez López Alberto Yeriel
Hernández Angón Diego
Morales Valdeolivar Gael Octavio

Grupo:

3PM – 4PM HRS

Clave:

3a3A

Especialidad:

Ing. informática

Fecha De Entrega

30/05/2023

REPORTE DE PRACTICAS

UNIDAD 5

Introducción

El ordenamiento es un proceso fundamental en la programación y la ciencia de la computación. Consiste en organizar un conjunto de elementos en una secuencia particular de acuerdo con ciertos criterios predefinidos. Los algoritmos de ordenamiento son utilizados en una amplia variedad de aplicaciones, desde la clasificación de datos en bases de datos hasta la presentación ordenada de información en aplicaciones web.

Existen numerosos métodos de ordenamiento, cada uno con sus propias características, ventajas y desventajas. En esta introducción, exploraremos algunos de los métodos de ordenamiento más comunes.

Competencia especifica.

Específica(s):

Conoce, comprende y aplica los algoritmos de ordenamiento para el uso adecuado en el desarrollo de aplicaciones que permita solucionar problemas del entorno.

Genéricas:

- Habilidad para buscar y analizar información proveniente de fuentes diversas.
- La comprensión y manipulación de ideas y pensamientos.
- Metodologías para solución de problemas, organización del tiempo y para el aprendizaje.
- Habilidad en el manejo de equipo de cómputo
- Capacidad para trabajar en equipo.
- Capacidad de aplicar los conocimientos en la práctica.

Marco teórico

5.1 Algoritmos de ordenamiento internos

Burbuja con señal

Análisis de eficiencia

El método de ordenamiento burbuja con señal, como mencionamos anteriormente, es una variante del algoritmo de ordenamiento burbuja que utiliza una señal para mejorar su eficiencia. La señal permite que el algoritmo se detenga antes si la lista ya está ordenada, evitando así comparaciones y cambios innecesarios.

En términos de complejidad de tiempo, el peor caso del algoritmo de ordenamiento burbuja con señal sigue siendo O(n^2), al igual que el algoritmo de ordenamiento burbuja tradicional. Esto se debe a que en el peor caso, donde la lista está en orden inverso, el algoritmo aún debe realizar todas las comparaciones y cambios necesarios para ordenar la lista correctamente.

Sin embargo, en promedio, la burbuja con señal puede ser más eficiente que el algoritmo de burbuja tradicional en casos donde la lista ya está parcialmente ordenada o cuando se alcanza una ordenación completa antes de recorrer todos los elementos. En estos casos, la señal permite que el algoritmo se detenga antes, reduciendo el número de comparaciones y cambios necesarios.

Es importante tener en cuenta que la mejora en la eficiencia de la burbuja con señal depende de la distribución de los elementos en la lista y de la posición en la que se encuentren los elementos desordenados. Si la lista está completamente desordenada, el beneficio de la señal puede ser mínimo.

En cuanto a la complejidad de espacio, tanto el algoritmo de burbuja tradicional como el de burbuja con señal tienen una complejidad de espacio de O(1) porque no requieren espacio adicional proporcional al tamaño de la lista.

En resumen, aunque la burbuja con señal puede mejorar el rendimiento promedio en comparación con el algoritmo de burbuja tradicional, su complejidad de tiempo sigue siendo O(n^2) en el peor caso. Por lo tanto, para listas grandes o conjuntos de datos complejos, puede ser más apropiado utilizar algoritmos de ordenamiento más eficientes, como Quicksort o Merge Sort, que tienen complejidades de tiempo más bajas, como O(n log n).

Análisis de los casos

1. **Mejor de los casos:**

El mejor de los casos ocurre cuando la lista ya está completamente ordenada. En este caso, el algoritmo de burbuja con señal tiene una ventaja significativa, ya que la señal evitará cualquier comparación o intercambio innecesario. La complejidad de tiempo en el mejor de los casos sería O(n) porque solo se requerirá una pasada para confirmar que la lista está ordenada. La complejidad espacial sigue siendo O(1) ya que no se requiere espacio adicional.

Ejemplo:

Consideremos una lista ordenada: [1, 2, 3, 4, 5].

El algoritmo de burbuja con señal realizará una pasada para verificar si se produce algún intercambio. Como la lista ya está ordenada, la señal permanecerá en falso y el algoritmo se detendrá. Por lo tanto, solo se realizará una comparación y no habrá intercambios.

2. **Caso medio:**

El caso medio se refiere a una distribución aleatoria de elementos en la lista. En este caso, no hay un orden particular en la lista y se requieren comparaciones y cambios para ordenarla. La complejidad de tiempo en el caso medio es O(n^2) porque, en promedio, el algoritmo requerirá comparar y cambiar elementos en cada pasada. La complejidad espacial sigue siendo O(1).

Ejemplo:

Consideremos una lista desordenada: [4, 2, 5, 1, 3].

El algoritmo de burbuja con señal realizará múltiples pasadas, comparando y cambiando elementos hasta que la lista esté completamente ordenada. En este caso, se requerirán varias comparaciones y cambios para ordenar la lista.

3. **Peor de los casos:**

El peor de los casos ocurre cuando la lista está ordenada en orden inverso. En este escenario, el algoritmo de burbuja con señal debe realizar comparaciones y cambios para cada par de elementos, lo que resulta en el peor rendimiento. La complejidad de tiempo en el peor de los casos sigue siendo O(n^2), ya que se deben realizar comparaciones y cambios en todas las pasadas. La complejidad espacial sigue siendo O(1).

Ejemplo:

Consideremos una lista ordenada en orden inverso: [5, 4, 3, 2, 1].

El algoritmo de burbuja con señal realizará múltiples pasadas, comparando y cambiando elementos en cada iteración hasta que la lista esté completamente ordenada. En este caso, se requerirán muchas comparaciones y cambios para ordenar la lista.

```java
@Override
public void burbujaSeñal() {
    boolean señal;
    int iteraciones=0;
    for (int i = 0; i < datos.length - 1; i++) {
        señal = false;
        for (int j = 0; j < datos.length - i - 1; j++) {
            if (datos[j] > datos[j + 1]) {
                // Intercambiar elementos datos[j] y datos[j + 1]
                int temp = datos[j];
                datos[j] = datos[j + 1];
                datos[j + 1] = temp;
                señal = true;
                iteraciones++;
            }
        }
        if (!señal)
            break;
    }
    System.out.println("Cantidad de ciclos: + " +iteraciones);
}
```

En resumen, la complejidad de tiempo del algoritmo de burbuja con señal sigue siendo O(n^2) en el peor caso y en el caso medio. Sin embargo, en el mejor de los casos, donde la lista ya está ordenada, el algoritmo puede tener una complejidad de tiempo de O(n). La complejidad espacial sigue siendo constante en todos los casos, es decir, O(1).

Doble burbuja

podemos considerar lo siguiente:

- Complejidad temporal: En el peor caso, cuando la lista está completamente desordenada, el algoritmo de doble burbuja realiza dos recorridos completos por la lista para cada iteración, lo que resulta en un tiempo de ejecución de O(n^2), donde "n" es el número de elementos en la lista. Esto es igual al tiempo de ejecución del algoritmo de burbuja clásico.

- Eficiencia: Aunque el método de doble burbuja no mejora la complejidad temporal del algoritmo de burbuja, puede mejorar la eficiencia en algunos casos al realizar recorridos en ambas direcciones. Esto permite que los elementos más grandes se

muevan hacia el final de la lista y los elementos más pequeños se muevan hacia el principio más rápidamente, acelerando el proceso de ordenamiento.

Sin embargo, es importante tener en cuenta que existen algoritmos de ordenamiento más eficientes, como el algoritmo de quicksort o el algoritmo de mergesort, que generalmente son preferibles para listas de gran tamaño debido a su mejor rendimiento en tiempo de ejecución.

```java
@Override
public void dobleBurbuja() {
    boolean señal;
    for (int i = 0; i < datos.length / 2; i++) {
        señal = false;
        for (int j = i; j < datos.length - i - 1; j++) {
            if (datos[j] > datos[j + 1]) {
                // Intercambiar elementos datos[j] y datos[j + 1]
                int temp = datos[j];
                datos[j] = datos[j + 1];
                datos[j + 1] = temp;
                señal = true;
            }
        }
        if (!señal) {
            break;
        }
        señal = 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;
                señal = true;
            }
        }
        if (!señal) {
            break;
        }
    }
}
```

Análisis de los casos

1. **Mejor de los casos:**

En el mejor de los casos, el arreglo ya se encuentra ordenado. En este escenario, el algoritmo realizará un recorrido ascendente por el arreglo y un recorrido descendente sin realizar ningún intercambio, ya que todos los elementos están en su posición correcta desde el principio. Por lo tanto, la complejidad temporal en el mejor caso es de O(n), donde “n” es el número de elementos en el arreglo. La complejidad espacial sigue siendo O(1) ya que no se requiere memoria adicional.

Ejemplo:

Supongamos que tenemos el arreglo [1, 2, 3, 4, 5]. Al ejecutar el algoritmo de doble burbuja en este arreglo, se realizará un recorrido ascendente y un recorrido

descendente sin realizar intercambios, ya que todos los elementos están en su posición correcta.

2. **Caso medio:**

En el caso medio, los elementos del arreglo están desordenados en algún grado aleatorio. El algoritmo realizará múltiples iteraciones, intercambiando los elementos necesarios hasta que el arreglo esté completamente ordenado. En promedio, el algoritmo realizará alrededor de n/2 iteraciones en cada recorrido, donde “n” es el número de elementos en el arreglo. Esto se debe a que los elementos más grandes y más pequeños tienden a moverse hacia las posiciones correctas en cada iteración.

La complejidad temporal en el caso medio es O(n^2), al igual que en el peor caso. Esto se debe a que, aunque los elementos pueden moverse más rápidamente en comparación con el algoritmo de burbuja clásico, todavía se requieren múltiples recorridos para ordenar completamente el arreglo. La complejidad espacial sigue siendo O(1) ya que no se requiere memoria adicional.

Ejemplo:

Supongamos que tenemos el arreglo [5, 2, 1, 4, 3]. Al ejecutar el algoritmo de doble burbuja en este arreglo, se realizarán múltiples iteraciones de recorrido ascendente y descendente hasta que el arreglo esté ordenado completamente.

3. **Peor de los casos:**

En el peor de los casos, el arreglo está ordenado en orden inverso, lo que implica que se requiere el máximo número de iteraciones y de intercambios en cada recorrido. En este caso, el algoritmo realizará n-1 iteraciones en cada recorrido, lo que resulta en un total de (n-1) * 2 iteraciones en total.

La complejidad temporal en el peor caso es O(n^2), al igual que en el caso medio. Esto se debe a que el algoritmo necesita realizar múltiples recorridos y comparaciones para ordenar completamente el arreglo. La complejidad espacial sigue siendo O(1) ya que no se requiere memoria adicional.

Ejemplo:

Supongamos que tenemos el arreglo [5, 4, 3, 2, 1].

Shell (incrementos – decrementos)

El método de ordenamiento Shell, también conocido como ordenamiento por inserción con incrementos y decrementos, es una mejora del algoritmo de ordenamiento por inserción. A diferencia del método de ordenamiento por inserción estándar, que compara elementos adyacentes, el método de Shell compara elementos distantes entre sí mediante una serie de incrementos o decrementos.

El algoritmo de ordenamiento Shel¡”se basa en la idea de realizar múltiples pasadas de ordenamiento en la lista, reduciendo gradualmente la brecha entre los elementos que se comparan. En cada pasada, los elementos se comparan a una distancia determinada, llamada “brecha” o “incremento”, y se intercambian si es necesario para lograr una mejor ordenación.

El proceso de ordenamiento Shell se puede resumir en los siguientes pasos:

1. Seleccionar una secuencia de incrementos. Comúnmente se utiliza la secuencia de incrementos de Knuth, que se calcula mediante la fórmula h = 3h + 1, donde h es el valor inicial de la brecha. La secuencia de incrementos se genera hasta que el valor de h sea menor o igual al tamaño de la lista.

2. Para cada brecha en la secuencia de incrementos, se realiza un paso de ordenamiento por inserción. El paso de ordenamiento por inserción es similar al algoritmo de inserción estándar, pero en lugar de comparar elementos adyacentes, se comparan elementos distantes “brecha” posiciones.

3. Se repite el paso anterior con brechas más pequeñas en la secuencia de incrementos hasta que la brecha sea igual a 1. En la última pasada, se realiza un ordenamiento por inserción estándar para garantizar que todos los elementos estén en su posición correcta.

En términos de eficiencia, el método de ordenamiento Shell tiene un rendimiento mejorado en comparación con el algoritmo de ordenamiento por inserción estándar. Sin embargo, su análisis de eficiencia es más complejo debido a la variabilidad de las brechas utilizadas.

La eficiencia del método de ordenamiento Shell depende de la secuencia de incrementos utilizada. En general, se ha demostrado que la secuencia de incrementos de Knuth tiene un buen rendimiento. El peor caso conocido para el método de Shell es O(n^(4/3)), lo cual es una mejora significativa sobre el peor caso del ordenamiento por inserción estándar, que es O(n^2).

En la práctica, el método de ordenamiento Shell tiende a ser más rápido que los métodos de ordenamiento cuadráticos (como la burbuja y la selección), pero es superado por algoritmos de ordenamiento más eficientes como el ordenamiento rápido (quicksort) o el ordenamiento de mezcla (merge sort) en la mayoría de los casos.

En resumen, el método de ordenamiento Shell es una técnica de ordenamiento mejorada basada en el ordenamiento por inserción. Aunque su análisis de eficiencia es más complejo debido a la variabilidad de las brechas utilizadas, en general, ofrece un rendimiento mejorado en comparación con el ordenamiento por inserción estándar, pero es superado por algoritmos más eficientes en la mayoría de los casos.

```java
@Override
public void dobleBurbuja() {
    boolean señal;
    for (int i = 0; i < datos.length / 2; i++) {
        señal = false;
        for (int j = i; j < datos.length - i - 1; j++) {
            if (datos[j] > datos[j + 1]) {
                // Intercambiar elementos datos[j] y datos[j + 1]
                int temp = datos[j];
                datos[j] = datos[j + 1];
                datos[j + 1] = temp;
                señal = true;
            }
        }
        if (!señal) {
            break;
        }
        señal = 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;
                señal = true;
            }
        }
        if (!señal) {
            break;
        }
    }
}
```

Análisis de los casos

El análisis de los casos mejor, medio y peor del algoritmo de ordenamiento Shell se basa en la elección de la secuencia de incrementos utilizada. Diferentes secuencias de incrementos pueden tener un impacto significativo en el rendimiento del algoritmo. A continuación, analizaremos cada caso y proporcionaremos ejemplos, así como la complejidad temporal y espacial asociada.

1. **Mejor caso:**

El mejor caso ocurre cuando la lista ya está completamente ordenada. En este caso, el algoritmo de ordenamiento Shell realiza una pasada final de ordenamiento por inserción estándar, lo cual es relativamente eficiente. La complejidad temporal en el mejor caso es aproximadamente O(n), donde n es el tamaño de la lista. Esto se debe a que el algoritmo solo realiza una pasada final para garantizar que los

elementos estén en su posición correcta. La complejidad espacial es O(1) porque no se requiere memoria adicional.

Ejemplo:

Supongamos que tenemos la siguiente lista ordenada: [1, 2, 3, 4, 5, 6]. En el mejor caso, el algoritmo de ordenamiento Shell simplemente realizará una pasada final de ordenamiento por inserción estándar para confirmar que los elementos están en su lugar.

2. **Caso medio:**

El caso medio del algoritmo de ordenamiento Shell es más difícil de analizar, ya que depende de la secuencia de incrementos utilizada. En general, el caso medio del método de ordenamiento Shell se considera mejor que el caso promedio de otros algoritmos de ordenamiento cuadráticos, como el ordenamiento por inserción o el ordenamiento de burbuja. Sin embargo, en comparación con algoritmos más eficientes, como el ordenamiento rápido o el ordenamiento de mezcla, el caso medio del método de Shell puede ser menos eficiente. La complejidad temporal en el caso medio varía según la secuencia de incrementos utilizada y puede ser difícil de determinar exactamente. En términos generales, se estima que el caso medio tiene una complejidad temporal de alrededor de O(n^(3/2)) o mejor. La complejidad espacial es O(1), ya que no se requiere memoria adicional.

Ejemplo:

Supongamos que tenemos la siguiente lista desordenada: [5, 2, 8, 1, 9, 3]. El rendimiento del algoritmo de ordenamiento Shell en el caso medio dependerá de la secuencia de incrementos utilizada.

3. **Peor caso:**

El peor caso del algoritmo de ordenamiento Shell ocurre cuando la secuencia de incrementos elegida no es óptima para la lista de entrada. En este caso, el rendimiento del algoritmo puede degradarse y acercarse al rendimiento del ordenamiento por inserción estándar. La complejidad temporal en el peor caso es aproximadamente O(n^(4/3)), lo cual es una mejora significativa en comparación con el ordenamiento por inserción estándar, pero no tan eficiente como otros algoritmos más avanzados. La complejidad espacial es O(1), ya que no se requiere memoria adicional.

Ejemplo:

Supongamos que tenemos una lista inversamente ordenada: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]. Si utilizamos una secuencia de brechas como [5, 3, 1], el algoritmo de ordenamiento Shell realizará múltiples pasadas y movimientos para ordenar la lista, ya que los elementos más grandes estarán lejos de sus posiciones finales.

Selección Directa

también conocido como Selection Sort en inglés, es un algoritmo de ordenamiento simple que ordena una lista de elementos encontrando repetidamente el elemento mínimo y colocándolo al principio de la lista no ordenada. A continuación, analizaré su eficiencia y describiré los casos mejor, medio y peor. Luego, proporcione el código en Java para implementar el algoritmo y realice una prueba de escritorio con los datos proporcionados.

Analisis de eficiencia:

Complejidad en el tiempo: El algoritmo de Selección Directa tiene una complejidad temporal de O(n^2), donde "n" es el número de elementos en la lista. Esto se debe

a que el algoritmo requiere dos bucles anidados. En cada iteración del bucle externo, se realiza una búsqueda del mínimo en el resto de la lista (bucle interno), lo cual requiere comparar cada elemento. Esto lleva a realizar aproximadamente (n-1) + (n-2) + ... + 1 comparaciones, lo que se reduce a (n * (n-1)) / 2 comparaciones en total.

Complejidad espacial: El algoritmo de Selección Directa tiene una complejidad espacial de O(1) porque no requiere estructuras de datos adicionales para ordenar la lista. El ordenamiento se realiza directamente en la lista de entrada, sin necesidad de memoria adicional.

Casos de mejor, medio y peor de los casos:

Mejor caso: El mejor caso ocurre cuando la lista ya está ordenada. En este caso, el algoritmo aún tiene que realizar las comparaciones para confirmar que la lista está ordenada, pero no es necesario realizar ningún intercambio. Aunque el número de comparaciones sigue siendo el mismo, la cantidad de intercambios es cero. Por lo tanto, en el mejor caso, la complejidad temporal sigue siendo O(n^2), pero puede ser más eficiente en términos de tiempo en comparación con otros algoritmos de ordenamiento.

Caso medio: El caso medio ocurre cuando la lista contiene elementos desordenados en diferentes posiciones. El algoritmo realiza comparaciones y swaps en cada iteración para encontrar y colocar el elemento mínimo en su posición correcta. La cantidad de comparaciones y swaps necesarios depende de la distribución de los elementos en la lista y puede variar.

Peor caso: El peor caso ocurre cuando la lista está ordenada en orden inverso. En este caso, se requiere la cantidad máxima de comparaciones y swaps en cada iteración para llevar el elemento mínimo al principio de la lista. Esto implica realizar aproximadamente (n * (n-1)) / 2 comparaciones y (n-1) swaps en total. Por lo tanto, en el peor caso,

```java
@Override
public void seleDirecta() {
    for (int i = 0; i < datos.length - 1; i++) {
        int indiceMinimo = i;
        for (int j = i + 1; j < datos.length; j++) {
            if (datos[j] < datos[indiceMinimo]) {
                indiceMinimo = j;
            }
        }
        int temp = datos[indiceMinimo];
        datos[indiceMinimo] = datos[i];
        datos[i] = temp;
    }
}
```

la complejidad temporal sigue siendo O(n^2).

Código en Java para implementar el algoritmo de Selección Directa:

Lista original: 23 -56 67 2 9 44 6 18 1 7

Lista ordenada: -56 1 2 6 7 9 18 23 44 67

En cada iteración, el algoritmo encuentra el elemento mínimo y lo coloca en la posición correcta. La lista se va ordenando gradualmente hasta que todos los elementos estén en su lugar.

Inserción directa

El algoritmo de inserción directa, también conocido como "ordenamiento por inserción", es un algoritmo simple de ordenamiento que funciona de la siguiente manera:

- Comienza con una lista desordenada de elementos.
- Toma el primer elemento de la lista y lo considera como "ordenado".
- Toma el siguiente elemento de la lista y lo compara con los elementos ordenados.

- Inserta el elemento en la posición correcta dentro de la porción ordenada de la lista, desplazando los elementos mayores hacia la derecha.
- Repite los pasos 3 y 4 hasta que todos los elementos estén ordenados.

Análisis de eficiencia:

El análisis de eficiencia de un algoritmo se realiza en términos de la cantidad de operaciones que realiza en relación al tamaño de entrada.

Complejidad tiempo: El algoritmo de inserción directa tiene una complejidad temporal promedio y peor caso de O(n^2), donde "n" es el número de elementos en el arreglo. En el mejor caso, cuando el arreglo ya está ordenado, la complejidad es de O(n) ya que no es necesario hacer intercambios.

Complejidad espacio: El algoritmo de inserción directa tiene una complejidad espacial de O(1), ya que solo se requiere una cantidad constante de espacio adicional para almacenar variables auxiliares.

Mejor caso: El mejor caso ocurre cuando el arreglo ya está ordenado. En este caso, el algoritmo solo realiza una pasada sobre el arreglo para verificar que los elementos están en orden, por lo que la complejidad es lineal (O(n)). Utilizando los datos proporcionados, el arreglo ya está desordenado, por lo que no es aplicable el mejor caso en este ejemplo.

Caso medio: El caso medio ocurre cuando los elementos del arreglo están en un orden aleatorio. En este caso, el algoritmo requiere realizar comparaciones e intercambios para cada elemento, lo que resulta en una complejidad cuadrática (O(n^2)). Utilizando los datos proporcionados, el caso medio sería el escenario en el que se encuentran los números desordenados.

Peor caso: El peor caso ocurre cuando el arreglo está ordenado en orden descendente. En este caso, el algoritmo realiza el máximo número de comparaciones e intercambios para cada elemento, lo que también resulta en una

complejidad cuadrática (O(n^2)). Utilizando los datos proporcionados, el peor caso sería si los números están ordenados en orden descendente.

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

Binario

El algoritmo de ordenamiento binario, también conocido como "ordenamiento por inserción binaria", es un algoritmo de ordenamiento eficiente que utiliza una estrategia de búsqueda binaria para encontrar la posición correcta de cada elemento en la lista durante el proceso de inserción.

A diferencia del ordenamiento por inserción estándar, que realiza comparaciones lineales para encontrar la posición adecuada, el ordenamiento binario reduce la cantidad de comparaciones realizadas utilizando la búsqueda binaria. Esto lo logra dividiendo la lista ordenada en dos partes y realizando comparaciones con el elemento central para determinar en qué mitad se debe realizar la búsqueda. Luego, repite este proceso en la mitad seleccionada hasta encontrar la posición correcta para insertar el elemento.

Descripción general del algoritmo de ordenamiento binario:

1. Comenzar con una lista desordenada de elementos.
2. Tomar el primer elemento de la lista y considerarlo como una lista ordenada de un solo elemento.
3. Para cada elemento restante en la lista desordenada:
    a. Realizar una búsqueda binaria en la lista ordenada para encontrar la posición correcta del elemento.
    b. Insertar el elemento en la posición encontrada en la lista ordenada.
4. Al finalizar, la lista estará completamente ordenada.

El algoritmo de ordenamiento binario tiene una complejidad temporal de O(n log n), donde n es el número de elementos en la lista. Esto se debe a que la búsqueda binaria se realiza para cada elemento, y la búsqueda binaria tiene una complejidad de O(log n). Sin embargo, el algoritmo de ordenamiento binario tiene una ventaja sobre el ordenamiento por inserción estándar, ya que reduce la cantidad de comparaciones realizadas, lo que lo hace más eficiente en la práctica.

ANALISIS DE EFICIENCIA

El algoritmo de ordenamiento binario es eficiente en términos de tiempo de ejecución debido a la estrategia de búsqueda binaria que utiliza. En lugar de realizar comparaciones lineales para encontrar la posición correcta de cada elemento, el algoritmo divide repetidamente la lista en dos partes y realiza comparaciones con el elemento central para determinar en qué mitad se debe realizar la búsqueda. Esto reduce significativamente el número de comparaciones necesarias y acelera el proceso de ordenamiento.

Además, el algoritmo de ordenamiento binario es capaz de ordenar la lista "en su lugar" (in-place), lo que significa que no requiere memoria adicional para realizar la

ordenación. Esto lo hace eficiente en términos de uso de memoria, especialmente cuando se trabaja con listas de gran tamaño o con recursos limitados.

El algoritmo de ordenamiento binario tiene un rendimiento notablemente mejor que los algoritmos de ordenamiento más simples, como el ordenamiento por inserción o el ordenamiento burbuja, en términos de tiempo de ejecución. Sin embargo, no es tan eficiente como los algoritmos más avanzados, como el ordenamiento rápido (quicksort) o el ordenamiento por fusión (merge sort), que tienen complejidades más bajas.

ANALISIS DE LOS CASOS

1. **Mejor caso:**

   El mejor caso ocurre cuando la lista de entrada ya está ordenada de manera ascendente. En este escenario, el algoritmo de ordenamiento binario tiene un rendimiento óptimo ya que no necesita realizar ninguna comparación adicional y simplemente recorre la lista una vez para confirmar que está ordenada correctamente. El número de comparaciones realizadas en el mejor caso es O(n), donde n es el número de elementos en la lista.

   Ejemplo:

   Supongamos que tenemos la siguiente lista: [2, 5, 7, 9, 12]. Como la lista ya está ordenada de manera ascendente, el algoritmo de ordenamiento binario no necesita realizar ninguna comparación adicional y la lista permanece sin cambios.

2. **Caso medio:**

   El caso medio ocurre cuando la lista de entrada no está preordenada y no sigue ningún patrón específico. En este caso, el algoritmo de ordenamiento binario realiza comparaciones y movimientos de elementos en diferentes posiciones de la lista para lograr la ordenación. La cantidad promedio de comparaciones realizadas en el caso medio es difícil de determinar con precisión y depende de la distribución de los elementos en la lista.

Ejemplo:

Supongamos que tenemos la siguiente lista desordenada: [5, 9, 2, 12, 7]. El algoritmo de ordenamiento binario realizará una serie de comparaciones y movimientos de elementos para ordenar la lista. Siguiendo el proceso del algoritmo, la lista se irá modificando gradualmente hasta que esté completamente ordenada: [2, 5, 7, 9, 12].

3. **Peor caso:**

   El peor caso ocurre cuando la lista de entrada está ordenada en orden descendente. En este caso, el algoritmo de ordenamiento binario realiza la máxima cantidad de comparaciones y movimientos de elementos para lograr la ordenación. La cantidad de comparaciones realizadas en el peor caso es O(n^2), donde n es el número de elementos en la lista.

   Ejemplo:

   Supongamos que tenemos la siguiente lista ordenada en orden descendente: [20, 15, 12, 10, 5]. El algoritmo de ordenamiento binario realizará múltiples comparaciones y movimientos de elementos para ordenar la lista. Siguiendo el proceso del algoritmo, la lista se irá modificando gradualmente hasta que esté completamente ordenada: [5, 10, 12, 15, 20].

COMPLEJIDAD EN EL TIEMPO

La complejidad temporal del algoritmo de ordenamiento binario es de O(n^2) en el peor caso y de O(n log n) en el mejor y caso promedio, donde n es el número de elementos en la lista a ordenar.

- Peor caso: En el peor caso, cuando la lista está ordenada en orden descendente, el algoritmo de ordenamiento binario realiza la máxima cantidad de comparaciones y movimientos de elementos para lograr la ordenación. La complejidad temporal en

el peor caso es cuadrática, O(n^2), ya que el algoritmo puede realizar hasta n comparaciones para cada elemento en la lista.

- Mejor y caso promedio: En el mejor y caso promedio, donde la lista no sigue un patrón específico, el algoritmo de ordenamiento binario utiliza la búsqueda binaria para reducir el número de comparaciones necesarias. Esto resulta en una complejidad temporal de O(n log n), donde n es el número de elementos en la lista. La búsqueda binaria tiene una complejidad logarítmica, y como se realiza para cada elemento en la lista, el algoritmo completo tiene una complejidad de O(n log n).

COMPLEJIDAD ESPACIAL

La complejidad espacial del algoritmo de ordenamiento binario es de O(1) en el caso del ordenamiento in situ (in-place), lo que significa que no requiere memoria adicional en relación al tamaño de la lista. El algoritmo ordena los elementos directamente en la lista de entrada sin necesidad de estructuras de datos adicionales. Esto hace que el algoritmo de ordenamiento binario sea eficiente en términos de uso de memoria, especialmente cuando se trabaja con listas de gran tamaño o con recursos limitados.

PRUEBA DE ESCRITORIO

Claro, realizaré una prueba de escritorio del algoritmo de ordenamiento binario utilizando los siguientes datos: 23, -56, 67, 2, 9, 44, 6, 18, 1, 7.

Paso 1:

Comenzamos con la lista desordenada: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7].

Paso 2:

Tomamos el primer elemento de la lista y lo consideramos como una lista ordenada de un solo elemento. En este caso, el primer elemento es 23.

Paso 3:

Para cada elemento restante en la lista desordenada, realizamos una búsqueda binaria en la lista ordenada para encontrar la posición correcta del elemento e insertarlo en esa posición.

- El segundo elemento es -56. Realizamos una búsqueda binaria en la lista ordenada [23] y encontramos que -56 debe ir antes de 23. Insertamos -56 en la lista ordenada, que ahora se convierte en [-56, 23].

- El tercer elemento es 67. Realizamos una búsqueda binaria en la lista ordenada [-56, 23] y encontramos que 67 debe ir después de -56 y antes de 23. Insertamos 67 en la lista ordenada, que ahora se convierte en [-56, 67, 23].

- Continuamos este proceso para los elementos restantes.

Al finalizar, obtenemos la lista completamente ordenada: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67].

IMPLEMENTACION DEL ALGORITMO CON DATOS ALEATORIOS

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

HeapSort

El algoritmo de ordenamiento Heap Sort es un algoritmo eficiente de ordenamiento basado en la estructura de datos conocida como montículo (heap). Consiste en dos fases principales: construcción del montículo y extracción del elemento máximo repetidamente para obtener una secuencia ordenada.

El algoritmo de ordenamiento Heap Sort se divide en dos partes:

1. Construcción del montículo (Heapify):
    - El primer paso del algoritmo es construir un montículo (heap) a partir del arreglo desordenado.
    - El montículo es una estructura de datos especial en forma de árbol binario completo, donde el valor de cada nodo es mayor o igual que los valores de sus hijos.
    - La construcción del montículo se realiza de abajo hacia arriba, comenzando desde el último nodo no hoja hasta la raíz.
    - Durante el proceso de construcción, se realiza una operación llamada "hundir" (sift-down) para colocar el elemento más grande en la posición correcta.
2. Extracción del elemento máximo:
    - Una vez que se ha construido el montículo, el elemento máximo (ubicado en la raíz) se intercambia con el último elemento del arreglo.
    - Luego, se reduce el tamaño del montículo en 1 y se realiza una operación llamada "hundir" (sift-down) en la raíz para restablecer la propiedad del montículo.
    - Este proceso se repite hasta que el montículo esté vacío y todos los elementos hayan sido extraídos y colocados en la secuencia ordenada.

Java proporciona implementaciones del algoritmo Heap Sort a través de los métodos `Arrays.sort()` y `PriorityQueue`. En el caso de `Arrays.sort()`, el algoritmo Dual-Pivot Quicksort se utiliza para arreglos de tipos primitivos, mientras que el algoritmo Timsort se utiliza para arreglos de objetos.

Implementación en las estructuras de datos.

```java
@Override
public void heapSort() {
    for (int i = datos.length / 2 - 1; i >= 0; i--) {
        heapify(datos, datos.length, i);
    }
    for (int i = datos.length - 1; i > 0; i--) {
        int temp = datos[0];
        datos[0] = datos[i];
        datos[i] = temp;
        heapify(datos, i, 0);
    }
}

public static void heapify(int[] datos, int length, int raiz) {
    int izq = 2 * raiz + 1;
    int der = 2 * raiz + 2;
    int mayor = raiz;
    if (izq < length && datos[izq] > datos[mayor]) {
        mayor = izq;
    }
    if (der < length && datos[der] > datos[mayor]) {
        mayor = der;
    }
    if (mayor != raiz) {
        int temp = datos[raiz];
        datos[raiz] = datos[mayor];
        datos[mayor] = temp;
        heapify(datos, length, mayor);
    }
}
```

Análisis de eficiencia:

El algoritmo de ordenamiento Heap Sort tiene una complejidad de tiempo de O(n log n) en todos los casos, y una complejidad de espacio constante de O(1). Estas características hacen de Heap Sort una opción eficiente para ordenar grandes conjuntos de datos, pero puede haber otros algoritmos que sean más eficientes en ciertos casos específicos.

Análisis de casos:

- **Mejor de los casos:**
  En el mejor de los casos, el arreglo de entrada ya está completamente ordenado o contiene elementos idénticos. En este escenario, el algoritmo de ordenamiento Heap Sort todavía debe realizar la construcción inicial del montículo, pero no se realizarán intercambios adicionales durante el proceso de extracción del elemento máximo. Por lo tanto, en el mejor de los casos, Heap Sort tiene una complejidad de tiempo de ejecución de O(n log n).

- **Caso medio:**

En el caso medio, se asume que los elementos del arreglo están distribuidos de manera aleatoria. En este escenario, Heap Sort realiza tanto la construcción del montículo como los intercambios durante el proceso de extracción del elemento máximo. La complejidad de tiempo promedio de Heap Sort en el caso medio también es de O(n log n).

- **Peor de los casos:**
  El peor de los casos ocurre cuando el arreglo de entrada está ordenado en orden inverso. En este escenario, Heap Sort requiere realizar la construcción completa del montículo en cada iteración, seguida de los intercambios durante la extracción del elemento máximo. La complejidad de tiempo en el peor de los casos para Heap Sort es O(n log n).

Complejidad en el tiempo.

El algoritmo de ordenamiento Heap Sort tiene una complejidad de tiempo promedio y peor caso de O(n log n), lo que lo hace eficiente para grandes conjuntos de datos. Sin embargo, su rendimiento no es tan bueno como otros algoritmos de ordenamiento, como Quick Sort o Merge Sort, en conjuntos de datos pequeños o en situaciones en las que el orden inicial del arreglo ya está cerca del orden final.

Complejidad en el espacio.

La complejidad en el espacio del algoritmo de ordenamiento Heap Sort es O(1) en su forma más común de implementación. Esto hace que Heap Sort sea una opción atractiva cuando se trabaja con conjuntos de datos grandes o cuando la disponibilidad de memoria es limitada.

En Heap Sort, el ordenamiento se realiza directamente en el arreglo de entrada, sin utilizar estructuras de datos adicionales. Solo se requiere un espacio constante para almacenar variables auxiliares y los índices de los elementos durante el proceso de construcción y extracción del montículo.

Esto significa que la cantidad de memoria utilizada por el algoritmo no depende del tamaño del arreglo de entrada, sino que es constante. No se crean arreglos adicionales ni se utilizan estructuras de datos auxiliares de tamaño proporcional al tamaño del arreglo original.

QuickSort (Recursivo)

El algoritmo QuickSort es un algoritmo de ordenamiento eficiente y ampliamente utilizado. Fue desarrollado por Tony Hoare en 1959 y se basa en la estrategia de "divide y vencerás". QuickSort es conocido por su rendimiento promedio rápido y su capacidad para manejar grandes conjuntos de datos.

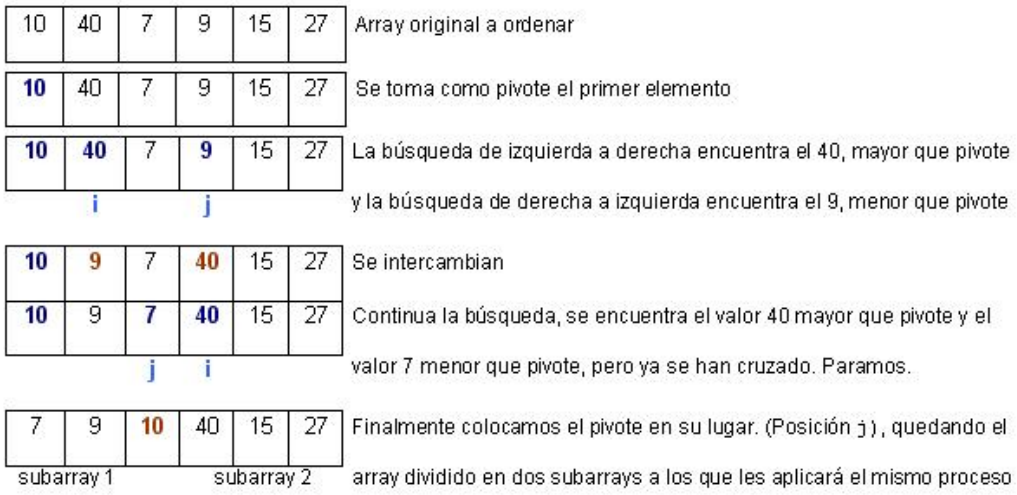
La idea central detrás de QuickSort es elegir un elemento del conjunto de datos, conocido como pivote, y luego dividir el conjunto en dos subconjuntos: uno con elementos menores que el pivote y otro con elementos mayores que el pivote. Luego, el algoritmo se aplica recursivamente a cada uno de los subconjuntos hasta que el arreglo esté completamente ordenado.

Proceso del algoritmo QuickSort:

- Elección del pivote: Selecciona un elemento del arreglo como pivote. El pivote puede ser cualquier elemento del arreglo, aunque comúnmente se elige el último elemento, el primero o uno aleatorio.
- Partición: Reorganiza el arreglo de manera que todos los elementos menores que el pivote se coloquen a su izquierda y los elementos mayores a su derecha. En esta etapa, el pivote se coloca en su posición final.
- Recursión: Aplica el algoritmo QuickSort de manera recursiva a los subconjuntos izquierdo y derecho del pivote. Esto implica repetir los pasos 1 y 2 para cada subconjunto hasta que todos los subconjuntos estén ordenados.

- Combinación: No es necesario realizar ninguna operación de combinación explícita, ya que los subconjuntos ya están ordenados in situ.

Gráficamente el proceso sería el siguiente:



Implementación en las estructuras de datos:

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

public static void QSR(int[] datos, int menor, int mayor) {
    if (menor < mayor) {
        int varIndex = particion(datos, menor, mayor);
        QSR(datos, menor, varIndex - 1);
        QSR(datos, varIndex + 1, mayor);
    }
}

public static int particion(int[] datos, int menor, int mayor) {
    int var = datos[mayor];
    int i = menor;
    for (int j = menor; j < mayor; j++) {
        if (datos[j] <= var) {
            int temp = datos[i];
            datos[i] = datos[j];
            datos[j] = temp;
            i++;
        }
    }
    int temp = datos[i];
    datos[i] = datos[mayor];
    datos[mayor] = temp;
    return i;
}
```

Análisis de eficiencia:

el algoritmo QuickSort tiene un rendimiento promedio rápido de O(n log n), lo que lo hace adecuado para ordenar grandes conjuntos de datos en la mayoría de los casos. Sin embargo, es importante tener en cuenta que su rendimiento puede degradarse hasta una complejidad cuadrática en el peor caso si no se toman medidas para evitar una partición desequilibrada.

Análisis de casos:

1. **Mejor de los casos:**

   En el mejor de los casos, cuando el pivote elegido divide el arreglo en dos subarreglos de tamaño similar en cada paso, el algoritmo QuickSort tiene una complejidad temporal de O(n log n).

   En este escenario ideal, el algoritmo realiza una partición equilibrada en cada nivel de recursión, lo que resulta en una altura del árbol de recursión de log

n. Por lo tanto, el número total de operaciones requeridas es proporcional a n log n.

2. **Caso promedio:**
   En el caso promedio, el algoritmo QuickSort tiene una complejidad temporal esperada de O(n log n).
   Aunque el peor caso tiene una complejidad cuadrática, en la práctica, los casos peores son poco comunes debido a la elección aleatoria del pivote y a las estrategias de partición utilizadas en implementaciones optimizadas.

3. **Peor caso:**
   En el peor de los casos, cuando el pivote elegido no divide el arreglo de manera equilibrada, la complejidad temporal del algoritmo QuickSort puede ser de O(n^2).
   Esto ocurre cuando el arreglo ya está ordenado en orden ascendente o descendente. En cada nivel de recursión, el pivote puede ser el elemento más pequeño o más grande del subarreglo, lo que resulta en un solo subarreglo de tamaño n-1.
   El algoritmo tendría que realizar n-1 niveles de recursión, lo que implica un tiempo total de ejecución cuadrático.

Complejidad en el tiempo:

QuickSort tiene una complejidad temporal de O(n log n), lo que significa que el tiempo de ejecución crece de manera logarítmica en función del tamaño del conjunto de datos. Esto hace que QuickSort sea muy eficiente en la mayoría de los casos.

Sin embargo, cuando el arreglo ya está ordenado en orden ascendente o descendente y el pivote elegido siempre divide el arreglo de manera desequilibrada, la complejidad temporal es de O(n^2). Esto se debe a que, en cada nivel de recursión, solo se elimina un elemento del arreglo, lo que lleva a una ejecución cuadrática en función del tamaño del conjunto de datos. Es importante tener en

cuenta este peor caso, aunque es poco común en la práctica, ya que el rendimiento de QuickSort en el caso promedio sigue siendo muy bueno.

Complejidad en el espacio:

En todos los casos, la complejidad en el espacio es de O(log n) debido al uso de la pila de llamadas en la recursión.

El espacio requerido por QuickSort está determinado por la profundidad máxima del árbol de recursión, que es logarítmica en función del tamaño del conjunto de datos. Cada llamada recursiva agrega un nuevo marco a la pila de llamadas, y la profundidad máxima de la pila de llamadas es logarítmica en función del tamaño del conjunto de datos.

Radix

El algoritmo Radix sort, también conocido como ordenamiento por base, es un algoritmo de ordenamiento eficiente que clasifica los elementos de una lista o arreglo de enteros en función de sus dígitos.

Análisis de eficiencia:

El algoritmo Radix sort tiene una complejidad temporal de O(k * n), donde n es el número de elementos en el arreglo yk es el número de dígitos del elemento más grande. En términos espaciales, el algoritmo utiliza espacio adicional para los arreglos auxiliares, por lo que su complejidad espacial es O(n).

Casos de eficiencia:

Mejor caso: El mejor caso ocurre cuando todos los elementos tienen el mismo número de dígitos. En este caso, el algoritmo realiza el método de conteo una sola

vez para cada dígito, lo que resulta en una complejidad de tiempo de O(k * n), donde k es el número de dígitos.

Caso medio: En el caso medio, los elementos tienen diferentes números de dígitos. La complejidad de tiempo sigue siendo O(k * n), pero k puede ser mayor que en el mejor caso, ya que debe considerar los dígitos de los elementos más grandes.

Peor caso: El peor caso ocurre cuando todos los elementos tienen diferentes números de dígitos yk es el número de dígitos del elemento más grande. En este caso, el algoritmo realiza el ordenamiento de conteo para cada dígito, lo que resulta en una complejidad de tiempo de O(k * n).

```java
@Override
public void radix() {
    int max = getMax(datos);
    int exp = 1;
    while (max / exp > 0) {
        conteo(datos, exp);
        exp *= 10;
    }
}

public static void conteo(int[] datos, int exp) {
    int n = datos.length;
    int[] output = new int[n];
    int[] count = new int[10];
    for (int num : datos) {
        count[(num / exp) % 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] / exp) % 10] - 1] = datos[i];
        count[(datos[i] / exp) % 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;
}
```

5.2 Algoritmos de ordenamiento externos

Interrelación

Este algoritmo utiliza el método de interrelación conocido como "Ordenamiento de burbuja" para ordenar los elementos del arreglo de manera ascendente. El algoritmo compara elementos auxiliares y los intercambiados si están en el orden incorrecto, repitiendo este proceso hasta que todo el arreglo esté ordenado.{

Algoritmo:

```java
@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;
    }
}
```

análisis de eficiencia:

Complejidad en el tiempo: En el peor de los casos, el algoritmo de interrelación tiene una complejidad de tiempo de O(n^2), donde n es el número de elementos en el arreglo. Esto se debe a los dos bucles anidados, que comparan todos los pares posibles de elementos. Sin embargo, en el mejor de los casos, donde el arreglo ya está ordenado, la complejidad de tiempo puede ser O(n) porque no se realizan intercambios.

Complejidad espacial: El algoritmo de interrelación tiene una complejidad espacial de O(1), ya que solo se utiliza una cantidad constante de espacio adicional para almacenar las variables temporales utilizadas en los intercambios.

Casos de prueba:

Mejor caso: El mejor caso ocurre cuando el arreglo ya está completamente ordenado. En este caso, no se realizará ningún intercambio y el algoritmo

recorrerá el arreglo una vez. Por lo tanto, en nuestro ejemplo, el mejor caso sería {1, 2, 6, 7, 9, 18, 23, 44, 67, -56}.

Caso medio: El caso medio es difícil de determinar sin conocer la distribución de los datos de entrada. En general, el algoritmo de interrelación tiende a tener un rendimiento más lento que otros algoritmos de ordenamiento más eficientes, como **QuickSort o MergeSort.**

Peor caso: El peor caso ocurre cuando el arreglo está ordenado en orden descendente. En este caso, el algoritmo ocurrirá el máximo número de comparaciones e intercambios. Para nuestro ejemplo, el peor caso sería {-56, 1, 2, 6, 7, 9, 18, 23, 44, 67}.

Mezcla Directa

El algoritmo de mezcla directa (o merge sort en inglés) es un algoritmo eficiente para ordenar elementos en una lista. A continuación, le mostraré cómo implementar el algoritmo de mezcla directa en Java, así como el análisis de eficiencia y los casos de mejor, medio y peor de los casos.

Implementación del algoritmo de mezcla directa en Java:

```java
@Override
public void mezclaDirecta() {
    int n = datos.length;
    int[] temp = new int[n];
    for (int tamArr = 1; tamArr < n; tamArr *= 2) {
        for (int inicio = 0; inicio < n - tamArr; inicio += 2 * tamArr) {
            int medio = inicio + tamArr;
            int fin = Math.min(inicio + 2 * tamArr, n);
            combinar(datos, temp, inicio, medio, fin);
        }
    }
}
public static void combinar(int[] datos, int[] temp, int inicio, int medio, int fin) {
    int i = inicio;
    int j = medio;
    int k = inicio;
    while (i < medio && j < fin) {
        if (datos[i] <= datos[j]) {
            temp[k] = datos[i];
            i++;
        } else {
            temp[k] = datos[j];
            j++;
        }
        k++;
    }
    while (i < medio) {
        temp[k] = datos[i];
        i++;
        k++;
    }while (j < fin) {
        temp[k] = datos[j];
        j++;
        k++;
    }
    for (int x = inicio; x < fin; x++) {
        datos[x] = temp[x];
    }
}
```

Analisis de eficiencia:

Tiempo de ejecución: El algoritmo de mezcla directa tiene una complejidad temporal promedio y peor caso de O(n log n), donde n es el tamaño del arreglo a ordenar. Esto hace que sea eficiente incluso para arreglos grandes. En el mejor caso, el tiempo de ejecución sigue siendo O(n log n). La eficiencia de este algoritmo radica en su enfoque de dividir y conquistar, dividiendo el arreglo en subarreglos más pequeños y luego fusionándolos en orden.

Espacio requerido: El algoritmo de mezcla directa requiere espacio adicional para almacenar los subarreglos durante la fase de mezcla. La complejidad espacial es O(n), ya que se requiere un arreglo auxiliar del mismo tamaño que el arreglo original.

Casos de mejor, medio y peor de los casos:

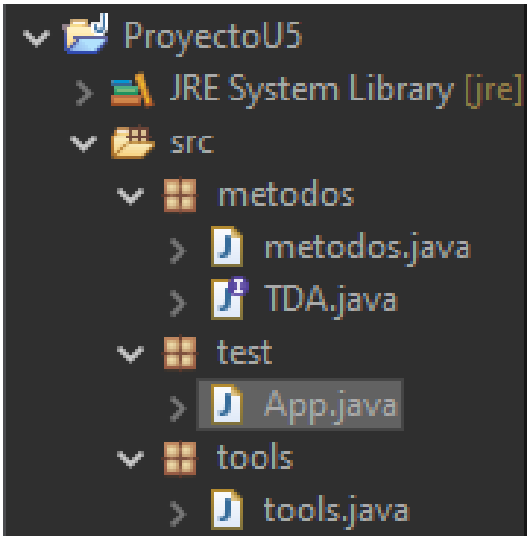
Mejor caso: El mejor caso ocurre cuando el arreglo ya está ordenado. En este caso, el algoritmo de mezcla directa todavía divide el arreglo en subarreglos, pero no necesita realizar ninguna operación de mezcla, ya que los subarreglos ya están ordenados. La complejidad temporal sigue siendo O(n log n).

Caso medio: El caso medio ocurre cuando los elementos del arreglo están desordenados de manera aleatoria. En este caso, el algoritmo divide recursivamente el arreglo en subarreglos y realiza la operación de mezcla para combinarlos en orden. La complejidad temporal sigue siendo O(n log n).

Peor caso: El peor caso ocurre cuando el arreglo está ordenado en orden inverso. En este caso, el algoritmo de mezcla directa divide el arreglo en subarreglos y realiza la operación de mezcla para combinarlos en orden. La complejidad temporal sigue siendo O(n log n), pero puede requerir más comparaciones y operaciones de mezcla que en los otros casos.

Desarrollo de la práctica

Para el presente reporte se creo la siguiente estructura de paquetes y clases para el correcto orden y ejecución del programa.



En el paquete de métodos se creo una clase llamada métodos en la que e encuentran todos los métodos

```java
package metodos;

public class metodos implements TDA {

    private byte idc;
    private byte p;
    int datos[];

    public metodos(byte idc){
        datos = new int[idc];
        p=-1;
        this.idc = idc;
    }

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

    @Override
    public boolean espacioArray() {
        return (p<=idc);
    }

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

    @Override
    public void almacenaDatos() {
        for(int i =0; i<datos.length;i++){
            datos[i] = generaRandom(10, 99);
            p++;
        }
    }

```

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

@Override
public void burbujaSeñal() {
    boolean señal;
    int iteraciones=0;
    for (int i = 0; i < datos.length - 1; i++) {
        señal = false;
        for (int j = 0; j < datos.length - i - 1; j++) {
            if (datos[j] > datos[j + 1]) {
                // Intercambiar elementos datos[j] y datos[j + 1]
                int temp = datos[j];
                datos[j] = datos[j + 1];
                datos[j + 1] = temp;
                señal = true;
                iteraciones++;
            }
        }
        if (!señal)
            break;
    }
    System.out.println("Cantidad de ciclos: + " +iteraciones);
}
```

```java
@Override
public void dobleBurbuja() {
    boolean señal;
    for (int i = 0; i < datos.length / 2; i++) {
        señal = false;
        for (int j = i; j < datos.length - i - 1; j++) {
            if (datos[j] > datos[j + 1]) {
                // Intercambiar elementos datos[j] y datos[j + 1]
                int temp = datos[j];
                datos[j] = datos[j + 1];
                datos[j + 1] = temp;
                señal = true;
            }
        }
        if (!señal) {
            break;
        }
        señal = 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;
                señal = true;
            }
        }
        if (!señal) {
            break;
        }
    }
}
```

```java
@Override
public void shellIncreDecre() {
    int incremento = datos.length / 2;

    while (incremento > 0) {
        for (int i = incremento; i < datos.length; i++) {
            int temp = datos[i];
            int j = i;
            while (j >= incremento && datos[j - incremento] > temp) {
                datos[j] = datos[j - incremento];
                j -= incremento;
            }
            datos[j] = temp;
        }
        incremento /= 2;
    }
}

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

```java
@Override
public void seleDirecta() {
    for (int i = 0; i < datos.length - 1; i++) {
        int indiceMinimo = i;
        for (int j = i + 1; j < datos.length; j++) {
            if (datos[j] < datos[indiceMinimo]) {
                indiceMinimo = j;
            }
        }
        int temp = datos[indiceMinimo];
        datos[indiceMinimo] = datos[i];
        datos[i] = temp;
    }
}

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

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

@Override
public void heapSort() {
    for (int i = datos.length / 2 - 1; i >= 0; i--) {
        heapify(datos, datos.length, i);
    }
    for (int i = datos.length - 1; i > 0; i--) {
        int temp = datos[0];
        datos[0] = datos[i];
        datos[i] = temp;
        heapify(datos, i, 0);
    }
}

public static void heapify(int[] datos, int length, int raiz) {
    int izq = 2 * raiz + 1;
    int der = 2 * raiz + 2;
    int mayor = raiz;
    if (izq < length && datos[izq] > datos[mayor]) {
        mayor = izq;
    }
    if (der < length && datos[der] > datos[mayor]) {
        mayor = der;
    }
    if (mayor != raiz) {
        int temp = datos[raiz];
        datos[raiz] = datos[mayor];
        datos[mayor] = temp;
        heapify(datos, length, mayor);
    }
}
```

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

    public static void QSR(int[] datos, int menor, int mayor) {
        if (menor < mayor) {
            int varIndex = particion(datos, menor, mayor);
            QSR(datos, menor, varIndex - 1);
            QSR(datos, varIndex + 1, mayor);
        }
    }

    public static int particion(int[] datos, int menor, int mayor) {
        int var = datos[mayor];
        int i = menor;
        for (int j = menor; j < mayor; j++) {
            if (datos[j] <= var) {
                int temp = datos[i];
                datos[i] = datos[j];
                datos[j] = temp;
                i++;
            }
        }
        int temp = datos[i];
        datos[i] = datos[mayor];
        datos[mayor] = temp;
        return i;
    }

    @Override
    public void radix() {
        int max = getMax(datos);
        int exp = 1;
        while (max / exp > 0) {
            conteo(datos, exp);
            exp *= 10;
        }
```

```java
    @Override
    public void radix() {
        int max = getMax(datos);
        int exp = 1;
        while (max / exp > 0) {
            conteo(datos, exp);
            exp *= 10;
        }
    }

    public static void conteo(int[] datos, int exp) {
        int n = datos.length;
        int[] output = new int[n];
        int[] count = new int[10];
        for (int num : datos) {
            count[(num / exp) % 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] / exp) % 10] - 1] = datos[i];
            count[(datos[i] / exp) % 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;
    }
```

```java
@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;
    }
}
```

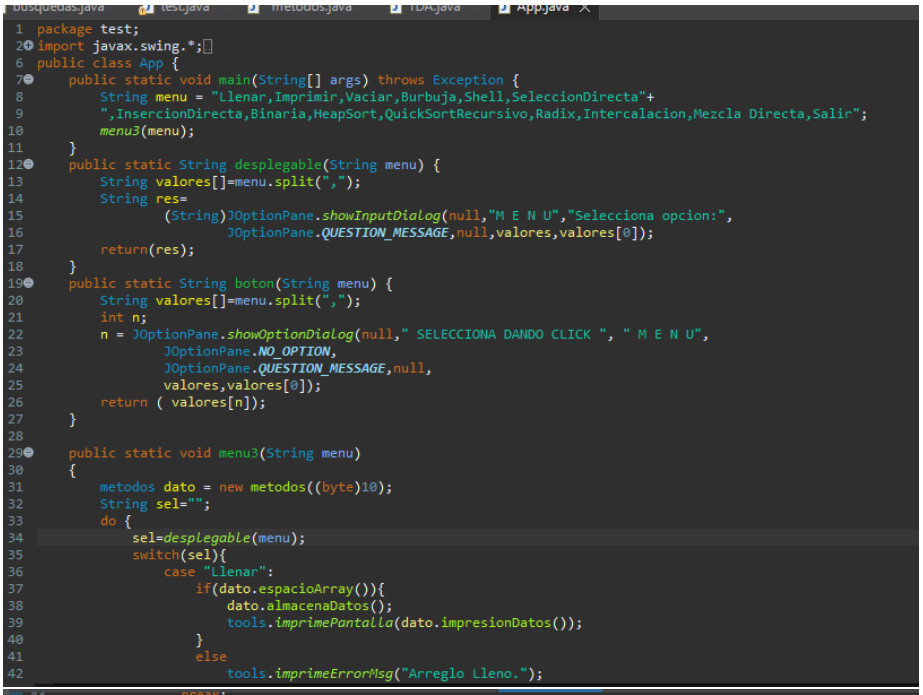
```java
    @Override
    public void mezclaDirecta() {
        int n = datos.length;
        int[] temp = new int[n];
        for (int tamArr = 1; tamArr < n; tamArr *= 2) {
            for (int inicio = 0; inicio < n - tamArr; inicio += 2 * tamArr) {
                int medio = inicio + tamArr;
                int fin = Math.min(inicio + 2 * tamArr, n);
                combinar(datos, temp, inicio, medio, fin);
            }
        }
    }

    public static void combinar(int[] datos, int[] temp, int inicio, int medio, int fin) {
        int i = inicio;
        int j = medio;
        int k = inicio;
        while (i < medio && j < fin) {
            if (datos[i] <= datos[j]) {
                temp[k] = datos[i];
                i++;
            } else {
                temp[k] = datos[j];
                j++;
            }
            k++;
        }
        while (i < medio) {
            temp[k] = datos[i];
            i++;
            k++;
        }
        while (j < fin) {
            temp[k] = datos[j];
            j++;
            k++;
        }
        for (int x = inicio; x < fin; x++) {
            datos[x] = temp[x];
        }
    }

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

Igual este paquete contiene una interfaz llamada TDA esta contiene

busquedas.java | test.java | *metodos.java | TDA.java | Ap

```java
package metodos;

public interface TDA {
    public boolean arrayVacio();
    public boolean espacioArray();
    public void vaciarArray();
    public void almacenaDatos();
    public String impresionDatos();
    public void burbujaSeñal();
    public void dobleBurbuja();
    public void shellIncreDecre();
    public void seleDirecta();
    public void inserDirecta();
    public void binaria();
    public void heapSort();
    public void quicksortRecursivo();
    public void radix();
    public void intercalacion();//mezcla simple
    public void mezclaDirecta();
    public int generaRandom(int min, int max);

}
```

En el otro paquete se llama test y esta contiene una clase que se llama App este contiene el menú

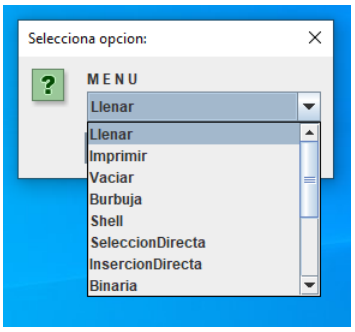


```java
package test;
import javax.swing.*;
public class App {
    public static void main(String[] args) throws Exception {
        String menu = "Llenar,Imprimir,Vaciar,Burbuja,Shell,SeleccionDirecta"+
        ",InsercionDirecta,Binaria,HeapSort,QuickSortRecursivo,Radix,Intercalacion,Mezcla Directa,Salir";
        menu3(menu);
    }
    public static String desplegable(String menu) {
        String valores[]=menu.split(",");
        String res=
                (String)JOptionPane.showInputDialog(null,"M E N U","Selecciona opcion:",
                        JOptionPane.QUESTION_MESSAGE,null,valores,valores[0]);
        return(res);
    }
    public static String boton(String menu) {
        String valores[]=menu.split(",");
        int n;
        n = JOptionPane.showOptionDialog(null," SELECCIONA DANDO CLICK ", " M E N U",
                JOptionPane.NO_OPTION,
                JOptionPane.QUESTION_MESSAGE,null,
                valores,valores[0]);
        return ( valores[n]);
    }

    public static void menu3(String menu)
    {
        metodos dato = new metodos((byte)10);
        String sel="";
        do {
            sel=desplegable(menu);
            switch(sel){
                case "Llenar":
                    if(dato.espacioArray()){
                        dato.almacenaDatos();
                        tools.imprimePantalla(dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Lleno.");
                break;
                case "Imprimir":
                    if(!dato.arrayVacio())
                        tools.imprimePantalla(dato.impresionDatos());
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "Vaciar":
                    if(!dato.arrayVacio()){
                        dato.vaciarArray();
                        tools.imprimePantalla("Arreglo vaciado.");
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "Burbuja":
                    if(!dato.arrayVacio()){
                        String s="";
                        do {
                            s=boton("BurbujaSeÃtal,BurbujaDoble,Salir");
                            switch(s){
                                case "BurbujaSeÃtal":
                                    tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                                    dato.burbujaSeñal();
                                    tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                                break;
                                case "BurbujaDoble":
                                    tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                                    dato.dobleBurbuja();
                                    tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                                break;
                                case "Salir":
                                break;
                            }
                        }while(!s.equalsIgnoreCase("Salir"));
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
```

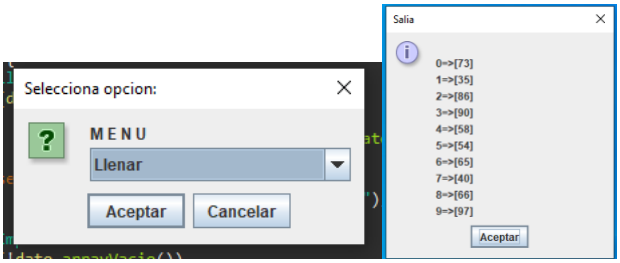
```
                    tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "Shell":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.shellIncreDecre();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "SeleccionDirecta":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.seleDirecta();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "InsercionDirecta":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.inserDirecta();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "Binaria":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.binaria();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                break;
                case "HeapSort":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.heapSort();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "QuickSortRecursivo":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.quicksortRecursivo();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "Radix":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.radix();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
                case "Intercalacion":
                    if(!dato.arrayVacio()){
                        tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                        dato.intercalacion();
                        tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                    }
                    else
                        tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
                break;
            case "Mezcla Directa":
                if(!dato.arrayVacio()){
                    tools.imprimePantalla("Arreglo desordenado:\n"+dato.impresionDatos());
                    dato.mezclaDirecta();
                    tools.imprimePantalla("Arreglo ordenado:\n"+dato.impresionDatos());
                }
                else
                    tools.imprimeErrorMsg("Arreglo Vacio.\nFavor de seleccionar LLenar.");
            break;
            case "Salir":

            break;
        }
    }while(!sel.equalsIgnoreCase("Salir"));
  }
}
```

Función de los métodos

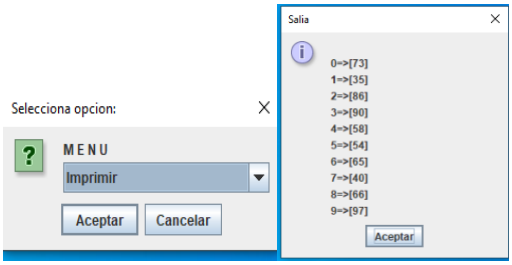
Cuando corres el programa se muestra un menú de todos los métodos



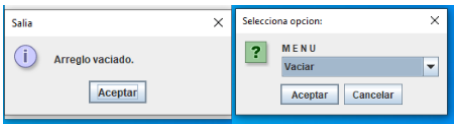
Seleccionamos llenar y muestra los valores



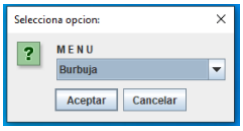
Si seleccionamos imprimir pues va a mostrar los datos ingresado



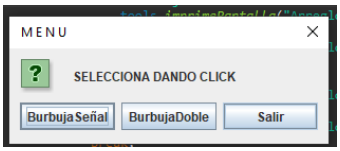
En la opción de vaciar pues mostrara un mensaje de arreglo vacío



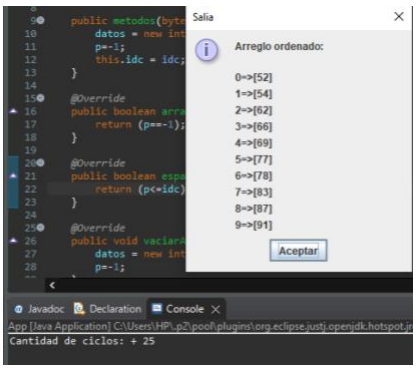
Método burbuja



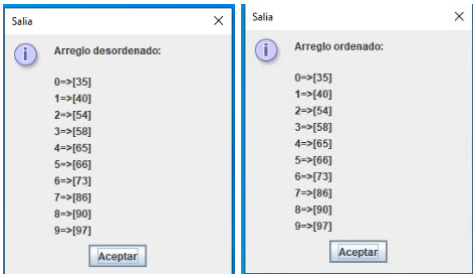
Si se selecciona mostrar otra opción donde deberás decir cual quieres si el de burbuja señal o el de burbuja doble



Seleccione burbuja señal y muestran los arreglos ordenados y desordenado

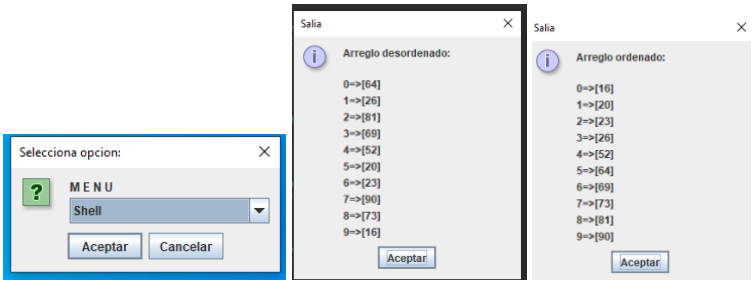


Selección burbuja doble y muestran los arreglos ordenados y desordenado

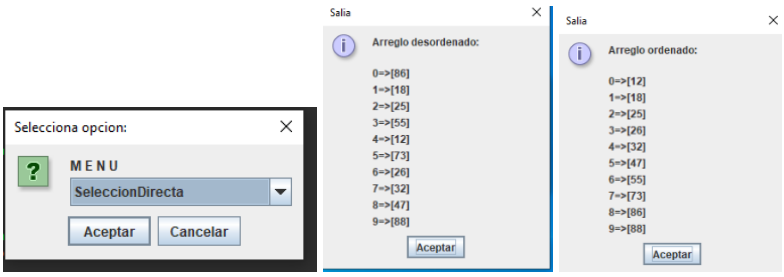


Metodo de shell

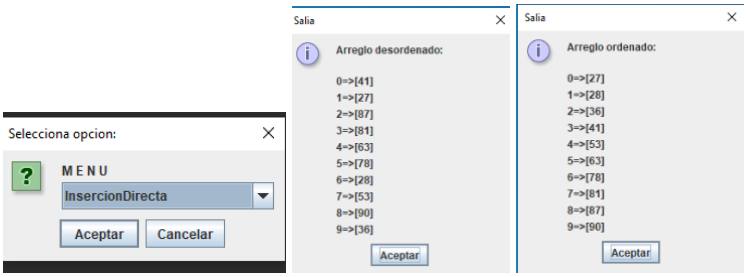
Este mustra el arreglo ordenado y desordenado



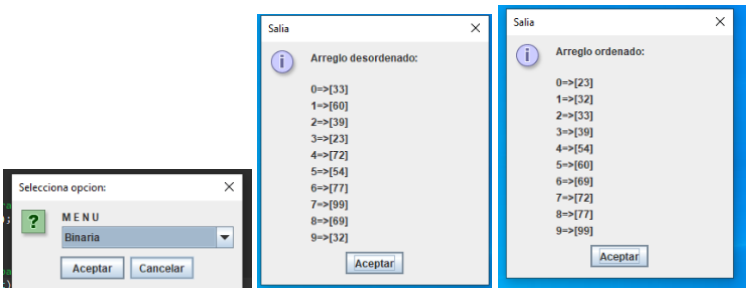
Selección Directa



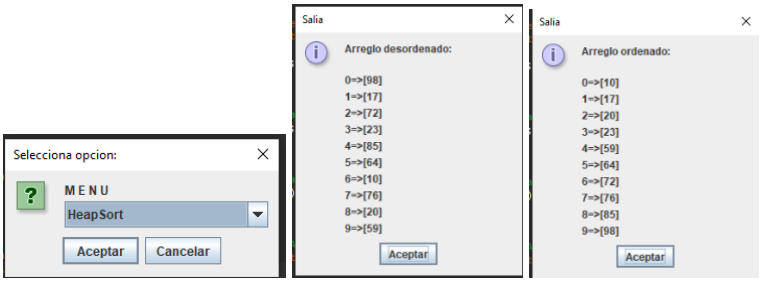
Inserccion Directa



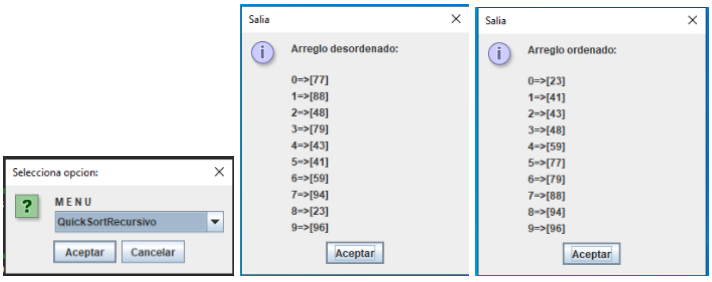
Binaria



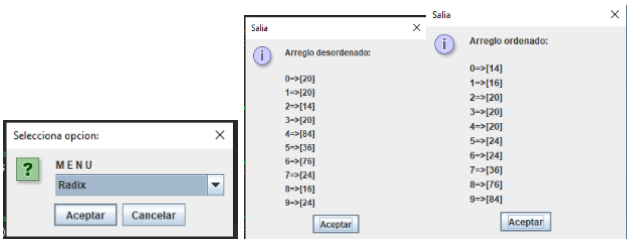
HeapSort



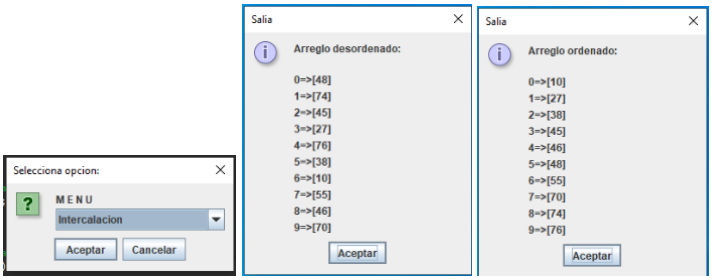
Quicksort recursivo



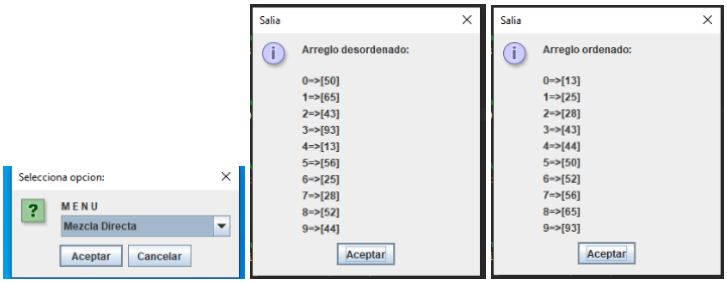
Radix



Intercalación



MezclaDirecta



Conclusión

existen varios métodos de ordenamiento que pueden utilizarse para organizar elementos en una lista de manera ascendente o descendente. Cada método tiene sus propias características y eficiencia, por lo que la elección del método necesita las necesidades adecuadas de las específicas del problema y de la cantidad de datos a ordenar.

Bibliografías

Específicos, O. (s/f). *Tema: Métodos de Ordenamiento. Parte 1* . Edu.sv. Recuperado el 30 de mayo de 2023, de https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-iv/2019/ii/guia-3.pdf