

Instituto Tecnológico De Orizaba

Estructura De Datos

Equipo 2

Integrantes:

Brayan Hernández Acevedo

Cesar Ríos Méndez

Pedro Senties Robles

Pedro Tezoco Cruz

Tema 6 Búsquedas Estructura de Datos

Fecha de entrega 31/05/2023



No.	Tema	Subtema-Re orde de Practicas
5	Métodos Ordenamientos	<p>Nºfarco Teórico deberá de incluir los subtemas investigados:</p> <p>5. I Algoritmos de ordenamiento internos</p> <p>Burbuja con señal</p> <p>Doble burbuja con señal</p> <p>Shell (incrementos - decrementos)</p> <p>Selección directa Inserción directa</p> <p>Binaria</p> <p>HeapSort</p> <p>Quicksoft recursivo</p> <p>5.2 Algoritmos de ordenamiento externos</p> <p>5.2.1 Intercalación</p> <p>5.2.2 Mezcla Directa</p> <p>5.2.3 Mezcla Natural</p>

Reporte de practica de los Algoritmos de Ordenamiento , deberá de investigar para \_\_\_\_\_ cada algoritmo: \_\_\_\_\_

- Análisis de eficiencia.
- Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos Complejidad en el tiempo y complejidad espacial.
- Prueba de escritorio por cada algoritmo, usando los siguientes datos:

23	-56	67	2	9	44	6	18	1	7
----	-----	----	---	---	----	---	----	---	---

- implementación de los algoritmos usando array unidimensional (memoria estática) y que almacene valores aleatorios.

Deberá de crear una interface que incluya:

```

public boolean arrayVacio() public
boolean espacioArray() public void
vaciarArray(); public void
almacenaAleatorios() public String
impresionDatos() public void
burbujaSeñal() public void
dobleBurbuja() public void
shellIncreDecre()

```



Orizaba



Instituto Tecnológico Nacional de  
TECNM México MEXICO Instituto Tecnológico de

```
public void seleDirecta() public void  
inserDirecta() public void binaria() public  
void heapSort() public void  
quicksortRecursivo() public void radix()  
public void intercalación() // mezcla simple  
public void mezclaDirecta() private int  
generaRandom (int min, int max)
```

### Recuerde:

Crear una clase que implemente la interface con 2 atributos privados:

- Un arreglo
- La variable índice de tipo byte, que almacena la cantidad de números aleatorios y la posición del arreglo donde se almacenará.

### Descripción de métodos

- El método constructor se encarga de crear el arreglo con n elementos e inicializa la variable índice con -1 (arreglo esta vacío).
- El método agregarDato llena el arreglo de números aleatorios de 2 cifras.
- métodos necesarios(implements)

Se anexa método para su implementación.

```
//Método privado private int generaRandom (int  
min, int max) { return (int)((max - min + 1) *  
Math.random()+ min);
```

Deberá de presentar un menú con las opciones necesarias, asegúrese que su

proyecto sea funcional.

Fecha de entrega: 30 Mayo 2023

Subir tanto proyectos y reporte en el repositorio usando en los anteriores temas

- **Análisis de eficiencia.**

### Algoritmos de ordenamiento internos

#### Burbuja Con Señal.

El método de ordenamiento de burbuja, también conocido como bubble sort en inglés, es un algoritmo de ordenamiento simple pero ineficiente. Su principio básico consiste en comparar pares adyacentes de elementos en una lista y intercambiarlos si están en el orden incorrecto. Este proceso se repite hasta que la lista esté completamente ordenada.

A continuación, proporcionaré un análisis de eficiencia del método de burbuja señalando sus características principales:

Complejidad temporal:

Mejor caso:  $O(n)$

Caso promedio:  $O(n^2)$

Peor caso:  $O(n^2)$

En el mejor caso, cuando la lista ya está ordenada, el algoritmo solo realiza una pasada para verificar que no haya intercambios, lo que implica una complejidad lineal de  $O(n)$ . Sin embargo, en el caso promedio y en el peor caso, donde la lista está desordenada, se necesitan dos bucles anidados para comparar y mover los elementos, lo que implica una complejidad cuadrática de  $O(n^2)$ . Esta ineficiencia se debe a que el algoritmo realiza comparaciones y posibles intercambios incluso cuando la lista ya está ordenada parcialmente.

Estabilidad:

El método de burbuja es un algoritmo estable, lo que significa que no cambia el orden relativo de elementos iguales. Si hay dos elementos iguales en la lista y uno está antes que el otro antes de la ordenación, entonces el elemento que está antes permanecerá antes después de la ordenación.

Uso de memoria:

El método de burbuja es un algoritmo in-place, lo que significa que no requiere memoria adicional para ordenar la lista. Las comparaciones y los intercambios se realizan directamente en la lista original.

En resumen, el método de burbuja es un algoritmo sencillo, pero poco eficiente en términos de tiempo de ejecución, especialmente para listas grandes. Su principal ventaja es que es fácil de implementar y entender. Sin embargo, para aplicaciones prácticas, se suelen preferir algoritmos de ordenamiento más eficientes, como el mergesort o el quicksort, que tienen complejidades temporales más bajas en promedio.

### Doble Burbuja Con Señal.

El método de ordenamiento de doble burbuja, también conocido como cocktail sort en inglés, es una variante del algoritmo de burbuja que busca mejorar su eficiencia al realizar pasadas de ordenamiento en ambas direcciones de la lista. Además, mencionas "con señal", lo cual no es un término estándar en relación al método de ordenamiento. Sin embargo, asumiré que te refieres a una señal o condición que indica si se han realizado intercambios en una pasada determinada.

A continuación, proporcionaré un análisis de eficiencia del método de doble burbuja señalando sus características principales:

Complejidad temporal:

Mejor caso:  $O(n)$

Caso promedio:  $O(n^2)$

Peor caso:  $O(n^2)$

Al igual que el algoritmo de burbuja, el método de doble burbuja tiene una complejidad cuadrática en los casos promedio y peor. Esto se debe a que se utilizan dos bucles anidados para comparar y mover los elementos en ambas direcciones de la lista. Sin embargo, en el mejor caso, cuando la lista ya está ordenada, el algoritmo puede detectarlo en la primera pasada y terminar, lo que resulta en una complejidad lineal de  $O(n)$ .

Estabilidad:

Al igual que el algoritmo de burbuja, el método de doble burbuja también es un algoritmo estable. No cambia el orden relativo de elementos iguales, lo que significa que si hay elementos iguales en la lista, su orden relativo se mantendrá después de la ordenación.

Uso de memoria:

Al igual que el algoritmo de burbuja, el método de doble burbuja es un algoritmo in-place. No requiere memoria adicional para ordenar la lista, ya que los intercambios y comparaciones se realizan directamente en la lista original.

En resumen, el método de doble burbuja es una variante del algoritmo de burbuja que busca mejorar la eficiencia al realizar pasadas de ordenamiento en ambas direcciones de la lista. Sin embargo, a pesar de esta mejora, sigue siendo un algoritmo con una complejidad cuadrática en la mayoría de los casos, lo que lo hace menos eficiente que otros algoritmos de ordenamiento más avanzados. Se recomienda utilizar algoritmos como el mergesort o el quicksort para aplicaciones prácticas donde se requiera un ordenamiento eficiente.

### Shell (incrementos - decrementos)

El método de ordenamiento Shell, también conocido como Shell sort, es un algoritmo de ordenamiento que combina la idea de dividir y conquistar con el método de inserción directa. La principal característica del método de Shell es que utiliza una secuencia de incrementos (también conocidos como brechas) para realizar comparaciones y movimientos en la lista de elementos. Estos incrementos se van reduciendo gradualmente hasta llegar a un incremento final de 1, momento en el cual se realiza una pasada final utilizando el método de inserción directa.

A continuación, proporcionaré un análisis de eficiencia del método de Shell (incrementos - decrementos):

Complejidad temporal:

Peor caso:  $O(n^2)$  a  $O(n \log^2 n)$

Caso promedio: depende de la secuencia de incrementos utilizada

La complejidad temporal del método de Shell depende en gran medida de la secuencia de incrementos utilizada. En el peor caso, cuando se utiliza una secuencia de incrementos desfavorable, el algoritmo puede tener una complejidad cuadrática de  $O(n^2)$ . Sin embargo, se han propuesto diferentes secuencias de incrementos que mejoran significativamente la eficiencia promedio del algoritmo, llegando a alcanzar una complejidad de  $O(n \log^2 n)$ .

La elección de una buena secuencia de incrementos es crucial para obtener un rendimiento óptimo del método de Shell. Algunas secuencias populares incluyen la secuencia de incrementos de Shell original ( $n/2, n/4, \dots, 1$ ), la secuencia de incrementos de Knuth ( $1, 4, 13, 40, \dots$ ), entre otras.

Estabilidad:

El método de Shell no es un algoritmo estable en su forma básica. Durante las pasadas de ordenamiento, los elementos pueden cambiar su orden relativo, lo que puede resultar en una falta de estabilidad en la ordenación final.

Uso de memoria:

El método de Shell es un algoritmo in-place, lo que significa que opera directamente sobre la lista original sin requerir memoria adicional para el ordenamiento.

En resumen, el método de Shell es un algoritmo de ordenamiento que utiliza una secuencia de incrementos para realizar comparaciones y movimientos en la lista. Aunque puede tener una complejidad cuadrática en el peor caso, se pueden utilizar secuencias de incrementos adecuadas para mejorar significativamente su

eficiencia promedio. Sin embargo, es importante tener en cuenta que el método de Shell no es estable en su forma básica y puede requerir adaptaciones adicionales para lograr estabilidad en la ordenación final.

### Selección directa Inserción directa

no ordenada y se intercambia con el elemento en la posición actual de la parte ordenada. Aunque el método de selección directa tiene una lógica simple de implementar, su eficiencia es baja debido a que siempre requiere una cantidad cuadrática de comparaciones e intercambios, sin importar si la lista ya está parcialmente ordenada o no.

Método de inserción directa:

Complejidad temporal:

Mejor caso:  $O(n)$

Caso promedio:  $O(n^2)$

Peor caso:  $O(n^2)$

En el método de inserción directa, se divide la lista en dos partes: una parte ordenada y una parte no ordenada. En cada iteración, se selecciona un elemento de la parte no ordenada y se inserta en la posición adecuada dentro de la parte ordenada. La eficiencia del método de inserción directa es mejor que la del método de selección directa en el mejor caso, donde la lista ya está ordenada. Sin embargo, en promedio y en el peor caso, ambos métodos tienen una complejidad cuadrática similar.

En resumen, tanto el método de selección directa como el método de inserción directa son algoritmos de ordenamiento simples pero ineficientes en términos de tiempo de ejecución. Ambos métodos tienen una complejidad cuadrática, lo que significa que pueden ser lentos para ordenar listas grandes. Para aplicaciones prácticas donde se requiere un ordenamiento eficiente, se recomienda utilizar algoritmos más avanzados como el mergesort o el quicksort, que tienen complejidades temporales más bajas en promedio.

### Binaria

El método de búsqueda binaria (binary search) es un algoritmo eficiente utilizado para buscar un elemento en una lista ordenada de manera ascendente o descendente. Aunque no es un algoritmo de ordenamiento propiamente dicho, es comúnmente utilizado en conjunto con algoritmos de ordenamiento para buscar elementos en listas ordenadas. A continuación, proporcionaré un análisis de eficiencia del método de búsqueda binaria:

Complejidad temporal:



Peor caso:  $O(\log n)$

Mejor caso:  $O(1)$

Caso promedio:  $O(\log n)$

La búsqueda binaria se basa en la división recursiva de la lista en mitades para buscar el elemento deseado. En cada iteración, se compara el elemento medio de la lista con el elemento buscado y se toma una decisión sobre qué mitad de la lista contiene el elemento. Este proceso continúa hasta que se encuentre el elemento deseado o se determine que no está presente en la lista.

La complejidad temporal de la búsqueda binaria es logarítmica en función del tamaño de la lista. En el peor caso, la búsqueda binaria requiere  $O(\log n)$  comparaciones para encontrar el elemento deseado. Esto se debe a que en cada iteración se divide la lista a la mitad, reduciendo rápidamente el espacio de búsqueda. En el mejor caso, cuando el elemento buscado se encuentra en la posición media de la lista, la búsqueda binaria puede encontrarlo en  $O(1)$  tiempo. En el caso promedio, también se requiere  $O(\log n)$  comparaciones para encontrar el elemento.

Requisitos de ordenamiento:

El método de búsqueda binaria requiere que la lista esté previamente ordenada en orden ascendente o descendente. Esto se debe a que el algoritmo depende de la propiedad de que los elementos en la lista estén ordenados para realizar comparaciones y dividir el espacio de búsqueda.

Uso de memoria:

La búsqueda binaria es un algoritmo que opera en lugar (in-place), lo que significa que no requiere memoria adicional más allá de la lista original y algunas variables adicionales para realizar las comparaciones y mantener el seguimiento de los índices.

En resumen, el método de búsqueda binaria es un algoritmo eficiente para buscar elementos en listas ordenadas. Su complejidad temporal logarítmica hace que sea una opción preferida en comparación con algoritmos de búsqueda lineal cuando se trata de buscar en grandes conjuntos de datos ordenados. Sin embargo, es importante destacar que la búsqueda binaria solo se puede aplicar en listas ordenadas y requiere un paso de ordenamiento previo.

## HeapSort

El método de ordenamiento HeapSort es un algoritmo de ordenamiento basado en la estructura de datos conocida como montículo o heap. Combina las propiedades de un montículo binario con una estrategia de ordenamiento para lograr una eficiencia promedio de  $O(n \log n)$ . A continuación, proporcionaré un análisis de eficiencia del método HeapSort:

Complejidad temporal:

Peor caso:  $O(n \log n)$

Caso promedio:  $O(n \log n)$

Mejor caso:  $O(n \log n)$

El HeapSort consiste en dos fases principales: la creación del montículo y la extracción de los elementos ordenados. La fase de creación del montículo requiere  $O(n)$  operaciones para convertir la lista desordenada en un montículo válido. Luego, en la fase de extracción, se extraen repetidamente los elementos del montículo y se reordena para obtener la lista ordenada final. La operación de extracción de un elemento del montículo y el reajuste para mantener la propiedad del montículo toma  $O(\log n)$  tiempo. Como se realizan estas operaciones para cada uno de los  $n$  elementos, la complejidad total es  $O(n \log n)$ .

Estabilidad:

El HeapSort no es inherentemente estable. Durante la fase de extracción, se intercambian los elementos del montículo para extraer el siguiente elemento máximo o mínimo, lo que puede alterar el orden relativo de elementos iguales. Sin embargo, es posible hacer que el HeapSort sea estable mediante técnicas adicionales, como agregar un índice adicional para mantener el orden original de elementos iguales.

Uso de memoria:

El HeapSort utiliza memoria adicional para almacenar el montículo binario. Esta memoria adicional es de tamaño  $O(n)$  debido a la construcción del montículo. Por lo tanto, el HeapSort no es un algoritmo in-place.

En resumen, el método HeapSort es un algoritmo de ordenamiento eficiente con una complejidad temporal de  $O(n \log n)$  en el peor caso y en promedio. Aunque no es un algoritmo in-place y requiere memoria adicional para el montículo, su eficiencia promedio lo convierte en una opción atractiva para ordenar grandes conjuntos de datos. Además, el HeapSort se beneficia de tener una estructura de datos eficiente para la prioridad y la selección de elementos, lo que lo hace útil en otras aplicaciones más allá del ordenamiento.

### Quicksoft recursivo

El método de ordenamiento QuickSort es un algoritmo de ordenamiento eficiente y ampliamente utilizado. Se basa en la técnica de "divide y vencerás" para dividir la lista en subconjuntos más pequeños, ordenarlos y combinarlos para obtener la lista ordenada final. A continuación, proporcionaré un análisis de eficiencia del método QuickSort recursivo:

Complejidad temporal:

Peor caso:  $O(n^2)$

Caso promedio:  $O(n \log n)$

Mejor caso:  $O(n \log n)$

El rendimiento del QuickSort depende de la elección del pivote y la distribución de los elementos en la lista. En el mejor caso y en el caso promedio, el QuickSort tiene una complejidad temporal de  $O(n \log n)$ . Esto se debe a que en cada iteración, el algoritmo divide la lista en dos partes aproximadamente iguales y realiza operaciones de partición en función del pivote seleccionado. La partición se realiza de tal manera que los elementos menores al pivote se encuentran en una sublista y los elementos mayores en otra. Este proceso continúa recursivamente hasta que se ordenen todos los subconjuntos y se combinen para obtener la lista ordenada final.

Sin embargo, en el peor caso, cuando el pivote elegido es siempre el elemento más grande o más pequeño, el QuickSort puede tener una complejidad cuadrática de  $O(n^2)$ . Esto ocurre cuando la lista ya está ordenada o casi ordenada, y en cada iteración el pivote divide la lista en dos partes muy desiguales. Para mitigar este problema, se pueden utilizar diferentes estrategias de elección del pivote, como seleccionar el pivote de forma aleatoria o utilizar el método de mediana de tres para obtener un rendimiento más consistente.

Estabilidad:

El QuickSort recursivo no es inherentemente estable. Durante el proceso de partición y ordenación de los subconjuntos, los elementos iguales pueden cambiar su orden relativo. Sin embargo, es posible hacer que el QuickSort sea estable mediante técnicas adicionales, como utilizar una comparación adicional para preservar el orden relativo de elementos iguales.

Uso de memoria:

El QuickSort recursivo es un algoritmo in-place, lo que significa que opera directamente sobre la lista original sin requerir memoria adicional para el ordenamiento. Sin embargo, la recursividad del algoritmo requiere el uso de la pila de llamadas, lo que puede consumir memoria adicional dependiendo del tamaño de la lista y la implementación específica.

En resumen, el método QuickSort recursivo es un algoritmo de ordenamiento eficiente con una complejidad promedio de  $O(n \log n)$ . Aunque puede tener un peor caso cuadrático, se pueden utilizar estrategias de elección de pivote y técnicas adicionales para mejorar su rendimiento. El QuickSort es ampliamente utilizado debido a su eficiencia y flexibilidad, y es especialmente útil para ordenar grandes conjuntos de datos.

## Algoritmos de ordenamiento externos

### Intercalación

El método de intercalación, también conocido como merge sort, es un algoritmo de ordenamiento eficiente que utiliza la estrategia de "dividir y conquistar" para ordenar una lista de elementos. A continuación, proporcionaré un análisis de eficiencia del método de intercalación:

Complejidad temporal:

Peor caso:  $O(n \log n)$

Caso promedio:  $O(n \log n)$

Mejor caso:  $O(n \log n)$

El método de intercalación divide la lista original en mitades hasta llegar a sublistas individuales. Luego, combina recursivamente las sublistas en orden ascendente o descendente utilizando la operación de intercalación. En cada nivel de recursión, el algoritmo realiza una intercalación de las sublistas, que requiere un tiempo proporcional a la suma de los elementos en las sublistas a combinar. Debido a la naturaleza del proceso de "dividir y conquistar", el método de intercalación tiene una complejidad temporal de  $O(n \log n)$  en todos los casos.

Estabilidad:

El método de intercalación es un algoritmo estable. Durante el proceso de intercalación, si dos elementos son iguales, se conserva su orden relativo en la lista ordenada final. Esto es posible porque las sublistas se combinan comparando los elementos de cada sublista de manera ordenada.

Uso de memoria:

El método de intercalación no es in-place, lo que significa que requiere memoria adicional para almacenar las sublistas temporales durante el proceso de intercalación. La cantidad de memoria requerida es proporcional al tamaño de la lista original. Esto puede ser una desventaja en comparación con algoritmos in-place como el QuickSort o el HeapSort, que no requieren memoria adicional. En resumen, el método de intercalación es un algoritmo de ordenamiento eficiente con una complejidad temporal de  $O(n \log n)$  en todos los casos. Es un algoritmo estable y adecuado para ordenar listas grandes, aunque requiere memoria adicional para almacenar las sublistas temporales durante el proceso de intercalación. El método de intercalación es ampliamente utilizado en la práctica debido a su eficiencia y estabilidad.

## Mezcla Directa

El método de ordenamiento Mezcla Directa (Direct Merge Sort) es un algoritmo de ordenamiento eficiente que utiliza la técnica de "dividir y combinar" para ordenar una lista de elementos. A continuación, proporcionaré un análisis de eficiencia del método de Mezcla Directa:

Complejidad temporal:

Peor caso:  $O(n \log n)$

Caso promedio:  $O(n \log n)$

Mejor caso:  $O(n \log n)$

En el método de Mezcla Directa, la lista se divide recursivamente en sublistas más pequeñas hasta que solo haya una lista con un solo elemento. Luego, las sublistas se combinan en orden utilizando la operación de mezcla (merge). Durante la fase de mezcla, los elementos de las sublistas se comparan y se colocan en el orden correcto para crear una lista ordenada final. La operación de mezcla tiene una complejidad temporal de  $O(n)$ , donde  $n$  es el tamaño total de las sublistas a combinar. Dado que en cada nivel de recursión se realizan operaciones de mezcla, la complejidad total del método de Mezcla Directa es de  $O(n \log n)$  en todos los casos.

Estabilidad:

El método de Mezcla Directa es un algoritmo estable. Durante la fase de mezcla, si dos elementos son iguales, se conserva su orden relativo en la lista ordenada final. Esto se logra al realizar las comparaciones y colocar los elementos en el orden correcto durante el proceso de mezcla.

Uso de memoria:

El método de Mezcla Directa puede ser implementado de dos formas: in-place y con uso de memoria adicional. La versión in-place modifica la lista original durante el proceso de mezcla, lo que significa que no requiere memoria adicional más allá de la lista original. Sin embargo, esta implementación puede tener una complejidad espacial mayor debido a la necesidad de realizar copias y desplazamientos de elementos. Por otro lado, la versión con uso de memoria adicional crea sublistas temporales durante el proceso de mezcla, lo que requiere memoria adicional proporcional al tamaño de la lista original.

En resumen, el método de Mezcla Directa es un algoritmo de ordenamiento eficiente con una complejidad temporal de  $O(n \log n)$  en todos los casos. Es un algoritmo estable y puede ser implementado de forma in-place o con uso de memoria adicional. La elección entre estas implementaciones depende de los

requisitos específicos de espacio y rendimiento. El método de Mezcla Directa es útil para ordenar listas grandes y es especialmente eficiente en aplicaciones donde se requiere estabilidad en el ordenamiento.

## Mezcla Natural

El método de ordenamiento Mezcla Natural (Natural Merge Sort) es un algoritmo de ordenamiento eficiente que combina los conceptos de Mezcla Directa y Ordenamiento por Mezcla para ordenar una lista de elementos. A continuación, proporcionaré un análisis de eficiencia del método de Mezcla Natural:

Complejidad temporal:

Peor caso:  $O(n^2)$

Caso promedio:  $O(n \log n)$

Mejor caso:  $O(n)$

El método de Mezcla Natural se basa en la idea de que una lista puede estar preordenada en múltiples secuencias ordenadas intercaladas. El algoritmo realiza iteraciones para encontrar estas secuencias ordenadas y luego las combina mediante la operación de mezcla. Durante la operación de mezcla, se compara y coloca en orden los elementos de las secuencias ordenadas para obtener una lista ordenada final.

En el peor caso, cuando la lista está inversamente ordenada, el algoritmo puede tener una complejidad cuadrática de  $O(n^2)$ . Esto ocurre cuando las secuencias ordenadas son pequeñas y se requieren muchas iteraciones para encontrarlas y combinarlas. Sin embargo, en el caso promedio y en el mejor caso, la complejidad temporal del Mezcla Natural es de  $O(n \log n)$ . Esto se debe a que el algoritmo aprovecha las secuencias ordenadas existentes para minimizar el número de comparaciones y operaciones de mezcla necesarias.

Estabilidad:

El método de Mezcla Natural es un algoritmo estable. Durante la operación de mezcla, si dos elementos son iguales, se conserva su orden relativo en la lista ordenada final. Esto se logra al realizar las comparaciones y colocar los elementos en el orden correcto durante el proceso de mezcla.

Uso de memoria:

El método de Mezcla Natural puede ser implementado de forma in-place o con uso de memoria adicional. La implementación in-place modifica la lista original durante el proceso de mezcla, lo que significa que no requiere memoria adicional más allá de la lista original. Sin embargo, esta implementación puede tener una complejidad



espacial mayor debido a la necesidad de realizar copias y desplazamientos de elementos. Por otro lado, la implementación con uso de memoria adicional crea sublistas temporales durante el proceso de mezcla, lo que requiere memoria adicional proporcional al tamaño de la lista original.

En resumen, el método de Mezcla Natural es un algoritmo de ordenamiento eficiente con una complejidad temporal de  $O(n \log n)$  en el caso promedio y en el mejor caso. Sin embargo, en el peor caso puede tener una complejidad cuadrática de  $O(n^2)$ . Es un algoritmo estable y puede ser implementado de forma in-place o con uso de memoria adicional. La elección entre estas implementaciones depende de los requisitos específicos de espacio y rendimiento. El método de Mezcla Natural es útil para ordenar listas grandes y aprovecha las secuencias ordenadas existentes para mejorar la eficiencia del ordenamiento.

**•Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos Complejidad en el tiempo y complejidad espacial.**

### Algoritmos de ordenamiento internos

#### Burbuja con señal.

El método de ordenamiento Burbuja con señal (Bubble Sort with Flag) es una variante del algoritmo Burbuja que utiliza una señal para mejorar su eficiencia al evitar iteraciones innecesarias cuando la lista ya está ordenada. A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del método de Burbuja con señal.

Mejor de los casos:

En el mejor de los casos, la lista de elementos ya se encuentra completamente ordenada. En este escenario, el algoritmo Burbuja con señal realiza una única pasada sobre la lista para verificar que no haya cambios y luego termina. Por lo tanto, la complejidad temporal en el mejor de los casos es  $O(n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que solo se realiza una comparación en cada iteración y no se requieren intercambios.

En cuanto a la complejidad espacial, el Burbuja con señal es un algoritmo in-place, lo que significa que no requiere memoria adicional más allá de la lista original. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(1)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

Se realiza una única pasada por la lista para verificar que no haya cambios.

El algoritmo termina.

Caso medio:

En el caso medio, el algoritmo Burbuja con señal realizará múltiples pasadas sobre la lista para ordenar los elementos. En cada pasada, se compara y se intercambian los elementos adyacentes según sea necesario hasta que la lista esté completamente ordenada. La cantidad de pasadas requeridas dependerá del número de elementos y de su disposición en la lista.

La complejidad temporal en el caso medio del Burbuja con señal es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que, en promedio, se requerirán comparaciones y posibles intercambios en cada elemento de la lista en cada pasada.

La complejidad espacial sigue siendo  $O(1)$  en el caso medio, ya que el algoritmo opera en la lista original sin utilizar memoria adicional.

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Pasada 1: [2, 4, 1, 3, 5]

Pasada 2: [2, 1, 3, 4, 5]

Pasada 3: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo Burbuja con señal requerirá el máximo número de pasadas y comparaciones para ordenar la lista. Esto ocurre cuando la lista está completamente desordenada, de manera que cada elemento debe ser intercambiado con todos los demás.

La complejidad temporal en el peor de los casos del Burbuja con señal es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que se requieren comparaciones y posibles intercambios en cada elemento de la lista en cada pasada.

Al igual que en los casos anteriores, la complejidad espacial es  $O(1)$  en el peor de los casos, ya que el algoritmo opera en la lista original sin utilizar memoria adicional.

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:



Pasada 1: [4, 3, 2, 1, 5]

Pasada 2: [3, 2, 1, 4, 5]

Pasada 3: [2, 1, 3, 4, 5]

Pasada 4: [1, 2, 3, 4, 5]

El algoritmo termina.

En resumen, el método Burbuja con señal tiene una complejidad temporal de  $O(n^2)$  en el peor de los casos y en el caso medio, y una complejidad temporal de  $O(n)$  en el mejor de los casos. La complejidad espacial es constante,  $O(1)$ , en todos los casos, ya que el algoritmo opera en la lista original sin utilizar memoria adicional. Es importante tener en cuenta que el Burbuja con señal puede ser menos eficiente que otros algoritmos de ordenamiento más avanzados, como el QuickSort o el MergeSort, especialmente para listas grandes.

### Doble burbuja con señal.

El método de ordenamiento Doble Burbuja con señal (Double Bubble Sort with Flag) es una variante del algoritmo Burbuja que utiliza dos indicadores de señal para mejorar su eficiencia al evitar iteraciones innecesarias cuando la lista ya está ordenada o casi ordenada. A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del método de Doble Burbuja con señal.

Mejor de los casos:

En el mejor de los casos, la lista de elementos ya se encuentra completamente ordenada. En este escenario, el algoritmo Doble Burbuja con señal realiza una única pasada sobre la lista para verificar que no haya cambios y luego termina. Por lo tanto, la complejidad temporal en el mejor de los casos es  $O(n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que solo se realiza una comparación en cada iteración y no se requieren intercambios.

En cuanto a la complejidad espacial, el Doble Burbuja con señal es un algoritmo in-place, lo que significa que no requiere memoria adicional más allá de la lista original. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(1)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

Se realiza una única pasada por la lista para verificar que no haya cambios.

El algoritmo termina.

Caso medio:

En el caso medio, el algoritmo Doble Burbuja con señal realizará múltiples pasadas sobre la lista para ordenar los elementos. En cada pasada, se compara y

se intercambian los elementos adyacentes según sea necesario hasta que la lista esté completamente ordenada. La cantidad de pasadas requeridas dependerá del número de elementos y de su disposición en la lista.

La complejidad temporal en el caso medio del Doble Burbuja con señal es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que, en promedio, se requerirán comparaciones y posibles intercambios en cada elemento de la lista en cada pasada.

La complejidad espacial sigue siendo  $O(1)$  en el caso medio, ya que el algoritmo opera en la lista original sin utilizar memoria adicional.

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Pasada 1: [2, 4, 1, 3, 5]

Pasada 2: [2, 1, 3, 4, 5]

Pasada 3: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo Doble Burbuja con señal requerirá el máximo número de pasadas y comparaciones para ordenar la lista. Esto ocurre cuando la lista está completamente desordenada, de manera que cada elemento debe ser intercambiado con todos los demás.

La complejidad temporal en el peor de los casos del Doble Burbuja con señal es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que se requieren comparaciones y posibles intercambios en cada elemento de la lista en cada pasada.

Al igual que en los casos anteriores, la complejidad espacial es  $O(1)$  en el peor de los casos, ya que el algoritmo opera en la lista original sin utilizar memoria adicional.

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

Pasada 1: [4, 3, 2, 1, 5]

Pasada 2: [3, 2, 1, 4, 5]

Pasada 3: [2, 1, 3, 4, 5]

Pasada 4: [1, 2, 3, 4, 5]

El algoritmo termina.

En resumen, el método Doble Burbuja con señal tiene una complejidad temporal de  $O(n^2)$  en el peor de los casos y en el caso medio, y una complejidad temporal de  $O(n)$  en el mejor de los casos. La complejidad espacial es constante,  $O(1)$ , en todos los casos, ya que el algoritmo opera en la lista original sin utilizar memoria adicional. Es importante tener en cuenta que el Doble Burbuja con señal puede ser menos eficiente que otros algoritmos de ordenamiento más avanzados, como el QuickSort o el MergeSort, especialmente para listas grandes.

### Shell (incrementos - decrementos)

El método de ordenamiento Shell (con incrementos y decrementos) es una variante del algoritmo de inserción directa que utiliza incrementos y decrementos para ordenar la lista de manera más eficiente. A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del método de Shell (incrementos - decrementos).

Mejor de los casos:

En el mejor de los casos, la lista de elementos ya se encuentra completamente ordenada. En este escenario, el algoritmo Shell (incrementos - decrementos) se comporta de manera similar al algoritmo de inserción directa, pero con la diferencia de que utiliza incrementos y decrementos para realizar las comparaciones y los desplazamientos de manera más eficiente. En este caso, la complejidad temporal es  $O(n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo solo realiza una pasada para verificar que la lista está ordenada y no necesita realizar intercambios.

En cuanto a la complejidad espacial, el algoritmo Shell (incrementos - decrementos) opera en la lista original sin requerir memoria adicional, por lo que su complejidad espacial en el mejor de los casos es  $O(1)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

Se realiza una pasada para verificar que la lista está ordenada.

El algoritmo termina.

Caso medio:

En el caso medio, el algoritmo Shell (incrementos - decrementos) realizará múltiples pasadas sobre la lista utilizando diferentes incrementos y decrementos. A medida que avanza, el algoritmo va reduciendo el tamaño de los incrementos y decrementos hasta llegar a un paso de 1, momento en el que se realiza una

pasada final utilizando el algoritmo de inserción directa. La cantidad de pasadas y el tamaño de los incrementos y decrementos dependen de la secuencia utilizada.

La complejidad temporal en el caso medio del algoritmo de Shell depende de la secuencia de incrementos y decrementos utilizada. En general, la complejidad temporal está entre  $O(n)$  y  $O(n^2)$ , siendo más cercana a  $O(n^2)$  en promedio. Sin embargo, debido a la variabilidad de las secuencias de incrementos y decrementos, es difícil establecer una complejidad precisa.

En cuanto a la complejidad espacial, el algoritmo Shell (incrementos - decrementos) opera en la lista original sin requerir memoria adicional, por lo que su complejidad espacial en el caso medio es  $O(1)$ .

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Se utilizan incrementos y decrementos para realizar múltiples pasadas sobre la lista.

Se realiza una pasada final utilizando el algoritmo de inserción directa.

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo Shell (incrementos - decrementos) requerirá el máximo número de pasadas y comparaciones para ordenar la lista. Esto ocurre cuando la lista está completamente desordenada y la secuencia de incrementos y decrementos no es óptima.

La complejidad temporal en el peor de los casos del algoritmo de Shell depende de la secuencia de incrementos y decrementos utilizada. En general, la complejidad temporal está entre  $O(n)$  y  $O(n^2)$ , siendo más cercana a  $O(n^2)$  en promedio. Sin embargo, debido a la variabilidad de las secuencias de incrementos y decrementos, es difícil establecer una complejidad precisa.

En cuanto a la complejidad espacial, el algoritmo Shell (incrementos - decrementos) opera en la lista original sin requerir memoria adicional, por lo que su complejidad espacial en el peor de los casos es  $O(1)$ .

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

Se utilizan incrementos y decrementos para realizar múltiples pasadas sobre la lista.

Se realiza una pasada final utilizando el algoritmo de inserción directa.  
El algoritmo termina.

En resumen, el método de Shell (incrementos - decrementos) tiene una complejidad temporal que depende de la secuencia de incrementos y decrementos utilizada, pero generalmente se encuentra entre  $O(n)$  y  $O(n^2)$ . La complejidad espacial es constante,  $O(1)$ , en todos los casos, ya que el algoritmo opera en la lista original sin requerir memoria adicional. Es importante tener en cuenta que la elección de la secuencia de incrementos y decrementos puede tener un impacto significativo en la eficiencia del algoritmo.

### Selección directa Inserción directa

El método de ordenamiento Selección directa-Inserción directa combina las estrategias de selección directa y inserción directa para ordenar una lista de elementos. A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del método Selección directa-Inserción directa.

Mejor de los casos:

En el mejor de los casos, la lista de elementos ya se encuentra completamente ordenada. En este escenario, el algoritmo de Selección directa-Inserción directa se comporta de manera similar al algoritmo de inserción directa. En cada pasada, se selecciona el elemento mínimo de la sublista no ordenada y se inserta en su posición correcta en la sublista ordenada. Sin embargo, en este caso, no se realizarán intercambios ya que los elementos ya están en su posición correcta.

La complejidad temporal en el mejor de los casos para el algoritmo Selección directa-Inserción directa es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que, aunque no se realicen intercambios, el algoritmo sigue iterando sobre la lista en cada pasada.

En cuanto a la complejidad espacial, el algoritmo Selección directa-Inserción directa opera en la lista original sin requerir memoria adicional, por lo que su complejidad espacial en el mejor de los casos es  $O(1)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

No se realizan intercambios ya que la lista está ordenada.

El algoritmo termina.

Caso medio:

En el caso medio, el algoritmo de Selección directa-Inserción directa realizará múltiples pasadas sobre la lista, seleccionando el elemento mínimo en cada pasada y moviéndolo a su posición correcta en la sublista ordenada. La cantidad de pasadas y comparaciones necesarias dependen del número de elementos y de su disposición en la lista.

La complejidad temporal en el caso medio del algoritmo Selección directa-Inserción directa es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que, en promedio, se requerirán comparaciones y desplazamientos en cada elemento de la lista en cada pasada.

En cuanto a la complejidad espacial, el algoritmo Selección directa-Inserción directa opera en la lista original sin requerir memoria adicional, por lo que su complejidad espacial en el caso medio es  $O(1)$ .

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Pasada 1: [1, 2, 4, 5, 3]

Pasada 2: [1, 2, 4, 5, 3]

Pasada 3: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo de Selección directa-Inserción directa requerirá el máximo número de pasadas y comparaciones para ordenar la lista. Esto ocurre cuando la lista está completamente desordenada.

La complejidad temporal en el peor de los casos para el algoritmo Selección directa-Inserción directa es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que se requerirán comparaciones y desplazamientos en cada elemento de la lista en cada pasada.

En cuanto a la complejidad espacial, el algoritmo Selección directa-Inserción directa opera en la lista original sin requerir memoria adicional, por lo que su complejidad espacial en el peor de los casos es  $O(1)$ .

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

Pasada 1: [1, 4, 3, 2, 5]

Pasada 2: [1, 2, 3, 4, 5]



El algoritmo termina.

En resumen, el método Selección directa-Inserción directa tiene una complejidad temporal de  $O(n^2)$  en el mejor de los casos, caso medio y peor de los casos. La complejidad espacial es constante,  $O(1)$ , en todos los casos, ya que el algoritmo opera en la lista original sin requerir memoria adicional. Es importante tener en cuenta que este método puede ser menos eficiente que otros algoritmos de ordenamiento más avanzados, como el QuickSort o el MergeSort, especialmente para listas grandes.

## Binaria

El método de búsqueda binaria es una técnica eficiente para encontrar un elemento en una lista ordenada. A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del método de búsqueda binaria.

Mejor de los casos:

En el mejor de los casos, el elemento buscado se encuentra exactamente en la mitad de la lista. En este escenario, la búsqueda binaria tiene un rendimiento óptimo, ya que el elemento objetivo se encuentra rápidamente.

La complejidad temporal en el mejor de los casos para la búsqueda binaria es  $O(1)$ , ya que se encuentra el elemento objetivo en la primera comparación.

En cuanto a la complejidad espacial, la búsqueda binaria no requiere memoria adicional más allá de la lista original, por lo que su complejidad espacial en el mejor de los casos es  $O(1)$ .

Ejemplo:

Lista ordenada: [1, 2, 3, 4, 5]

Elemento buscado: 3

Pasos:

Comparación 1: El elemento medio es 3, que coincide con el elemento buscado. El algoritmo termina.

Caso medio:

En el caso medio, el elemento buscado puede estar en cualquier posición de la lista ordenada. En cada paso, se divide la lista a la mitad y se compara el elemento medio con el objetivo.

La complejidad temporal en el caso medio para la búsqueda binaria es  $O(\log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que en cada paso la lista se divide a la mitad, reduciendo el espacio de búsqueda a la mitad en cada iteración.

En cuanto a la complejidad espacial, la búsqueda binaria no requiere memoria adicional más allá de la lista original, por lo que su complejidad espacial en el caso medio es  $O(1)$ .

Ejemplo:

Lista ordenada: [1, 2, 3, 4, 5]

Elemento buscado: 4

Pasos:

Comparación 1: El elemento medio es 3, se descarta la mitad izquierda.

Comparación 2: El elemento medio es 4, que coincide con el elemento buscado.

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el elemento buscado no se encuentra en la lista o se encuentra en la última posición de la lista. En cada paso, la lista se divide a la mitad y se descarta una mitad de la lista.

La complejidad temporal en el peor de los casos para la búsqueda binaria es  $O(\log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que en cada paso la lista se divide a la mitad, reduciendo el espacio de búsqueda a la mitad en cada iteración.

En cuanto a la complejidad espacial, la búsqueda binaria no requiere memoria adicional más allá de la lista original, por lo que su complejidad espacial en el peor de los casos es  $O(1)$ .

Ejemplo:

Lista ordenada: [1, 2, 3, 4, 5]

Elemento buscado: 6

Pasos:

Comparación 1: El elemento medio es 3, se descarta la mitad izquierda.

Comparación 2: El elemento medio es 5, se descarta la mitad derecha.

El algoritmo termina sin encontrar el elemento buscado.

En resumen, el método de búsqueda binaria tiene una complejidad temporal de  $O(\log n)$  en el caso medio y peor de los casos, y  $O(1)$  en el mejor de los casos. La complejidad espacial es constante,  $O(1)$ , en todos los casos, ya que el algoritmo opera en la lista original sin requerir memoria adicional. La búsqueda binaria es particularmente eficiente para listas grandes y ordenadas, ya que reduce el espacio de búsqueda a la mitad en cada iteración.



## HeapSort

El algoritmo HeapSort utiliza un heap binario para ordenar una lista de elementos. A continuación, se presenta un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del algoritmo HeapSort.

Mejor de los casos:

En el mejor de los casos, la lista de elementos ya está ordenada o casi ordenada. Sin embargo, HeapSort no puede aprovechar esta condición inicial y siempre requiere realizar una construcción completa del heap antes de realizar las operaciones de extracción. Por lo tanto, el mejor de los casos para HeapSort sigue teniendo una complejidad similar al caso medio y peor de los casos.

La complejidad temporal en el mejor de los casos para HeapSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que se debe construir un heap completo y luego realizar extracciones sucesivas para ordenar la lista.

En cuanto a la complejidad espacial, HeapSort requiere espacio adicional para el heap binario. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

Construcción del heap: [5, 4, 3, 2, 1]

Extracción sucesiva: [1, 2, 3, 4, 5]

El algoritmo termina.

Caso medio:

En el caso medio, los elementos de la lista están desordenados de manera aleatoria. HeapSort realiza una construcción del heap y luego realiza extracciones sucesivas para ordenar la lista.

La complejidad temporal en el caso medio para HeapSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que se debe construir un heap completo y luego realizar extracciones sucesivas para ordenar la lista.

En cuanto a la complejidad espacial, HeapSort requiere espacio adicional para el heap binario. Por lo tanto, la complejidad espacial en el caso medio es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Construcción del heap: [5, 3, 4, 1, 2]

Extracción sucesiva: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, los elementos de la lista están ordenados en orden descendente. Esto provoca que la construcción del heap sea más costosa y que se realicen más operaciones de intercambio.

La complejidad temporal en el peor de los casos para HeapSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que se debe construir un heap completo y luego realizar extracciones sucesivas para ordenar la lista.

En cuanto a la complejidad espacial, HeapSort requiere espacio adicional para el heap binario. Por lo tanto, la complejidad espacial en el peor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

Construcción del heap: [5, 4, 3, 2, 1]

Extracción sucesiva: [1, 2, 3, 4, 5]

El algoritmo termina.

En resumen, el método HeapSort tiene una complejidad temporal de  $O(n \log n)$  en el mejor de los casos, caso medio y peor de los casos. La complejidad espacial es  $O(n)$  en todos los casos, ya que requiere espacio adicional para el heap binario. Aunque HeapSort puede ser eficiente para listas grandes, su desventaja es que no es un algoritmo de ordenamiento estable y tiene una constante oculta mayor que otros algoritmos de ordenamiento como MergeSort o QuickSort.

### Quicksoft recursivo

Me disculpo, pero parece que ha habido un error tipográfico en la terminología utilizada. No existe un método de ordenamiento llamado "Quicksoft" en el contexto de algoritmos de ordenamiento. El método de ordenamiento comúnmente conocido y ampliamente utilizado es el algoritmo de ordenamiento rápido, también conocido como QuickSort.

El algoritmo de QuickSort es un algoritmo de ordenamiento basado en la técnica de "divide y vencerás". A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del algoritmo QuickSort.

Mejor de los casos:

En el mejor de los casos, el pivote elegido divide la lista en dos partes de igual tamaño (o aproximadamente igual). Esto significa que en cada paso de partición, el pivote divide la lista en dos sublistas de tamaño similar, lo que lleva a una partición equilibrada en cada nivel del árbol de recursión.

La complejidad temporal en el mejor de los casos para QuickSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que en cada nivel del árbol de recursión se realiza una partición en tiempo lineal, y el árbol de recursión tiene una altura de  $\log n$ .

En cuanto a la complejidad espacial, QuickSort utiliza la recursión y no requiere memoria adicional más allá de la lista original. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(\log n)$  debido a la pila de llamadas recursivas.

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

Elección del pivote: 3

Partición: [1, 2] [3] [4, 5]

Ordenamiento de sublistas: [1, 2] [3] [4, 5]

El algoritmo termina.

Caso medio:

En el caso medio, los elementos de la lista están desordenados de manera aleatoria. La elección del pivote puede variar y puede dividir la lista en sublistas de diferentes tamaños en cada nivel del árbol de recursión.

La complejidad temporal en el caso medio para QuickSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que, en promedio, el algoritmo divide la lista en dos sublistas de tamaño aproximadamente igual en cada nivel del árbol de recursión.

En cuanto a la complejidad espacial, QuickSort utiliza la recursión y no requiere memoria adicional más allá de la lista original. Por lo tanto, la complejidad espacial en el caso medio es  $O(\log n)$  debido a la pila de llamadas recursivas.

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Elección del pivote: 3

Partición: [2, 1] [3] [5, 4]

Ordenamiento de sublistas: [1, 2] [3] [4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el pivote elegido en cada paso divide la lista en dos sublistas de tamaño desigual, por ejemplo, el pivote siempre es el elemento más pequeño o el elemento más grande de la lista. Esto provoca una partición desequilibrada en cada nivel del árbol de recursión.

La complejidad temporal en el peor de los casos para QuickSort es  $O(n^2)$ , donde  $n$  es el número de elementos en la lista. Esto ocurre cuando la partición es siempre desequilibrada y se realiza en tiempo cuadrático en cada nivel del árbol de recursión.

En cuanto a la complejidad espacial, QuickSort utiliza la recursión y no requiere memoria adicional más allá de la lista original. Por lo tanto, la complejidad espacial en el peor de los casos es  $O(\log n)$  debido a la pila de llamadas recursivas.

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

Elección del pivote: 1

Partición: [1] [5, 4, 3, 2]

Ordenamiento de sublistas: [1] [2, 3, 4, 5]

El algoritmo termina.

En resumen, el algoritmo QuickSort tiene una complejidad temporal promedio de  $O(n \log n)$  en el caso medio y una complejidad temporal en el peor de los casos de  $O(n^2)$ . La complejidad espacial es  $O(\log n)$  en ambos casos debido a la recursión. Aunque QuickSort es un algoritmo eficiente en promedio, su peor caso puede ser costoso, especialmente si la lista ya está casi ordenada o está en orden inverso.

## Algoritmos de ordenamiento externos

### Intercalación

El método de ordenamiento por intercalación, también conocido como MergeSort, es un algoritmo de ordenamiento eficiente basado en la técnica de "divide y vencerás". A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del algoritmo de intercalación (MergeSort).

Mejor de los casos:

En el mejor de los casos, el algoritmo de intercalación ya se encuentra en un estado ordenado o casi ordenado. En este caso, el algoritmo simplemente realiza las comparaciones necesarias y fusiona las sublistas sin la necesidad de realizar intercambios adicionales.

La complejidad temporal en el mejor de los casos para MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo divide repetidamente la lista en sublistas de tamaño medio hasta que cada sublista contenga un solo elemento, y luego fusiona las sublistas en orden creciente.

En cuanto a la complejidad espacial, MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

División en sublistas: [1] [2] [3] [4] [5]

Fusión de sublistas: [1, 2, 3, 4, 5]

El algoritmo termina.

Caso medio:

En el caso medio, los elementos de la lista están desordenados de manera aleatoria. El algoritmo de intercalación divide repetidamente la lista en sublistas de tamaño medio y luego fusiona las sublistas en orden creciente.

La complejidad temporal en el caso medio para MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo divide repetidamente la lista en sublistas de tamaño medio hasta que cada sublista contenga un solo elemento, y luego fusiona las sublistas en orden creciente.

En cuanto a la complejidad espacial, MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el caso medio es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

División en sublistas: [5, 2] [4] [1, 3]

Fusión de sublistas: [2, 5] [4] [1, 3]

Fusión final: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo de intercalación debe realizar el máximo número de comparaciones y fusiones. Esto ocurre cuando la lista está en orden inverso.

La complejidad temporal en el peor de los casos para MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo

divide repetidamente la lista en sublistas de tamaño medio hasta que cada sublista contenga un solo elemento, y luego fusiona las sublistas en orden creciente.

En cuanto a la complejidad espacial, MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el peor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

División en sublistas: [5] [4] [3] [2] [1]

Fusión de sublistas: [4, 5] [2, 3] [1]

Fusión final: [2, 3, 4, 5, 1]

El algoritmo termina.

En resumen, el algoritmo de intercalación (MergeSort) tiene una complejidad temporal de  $O(n \log n)$  en el mejor de los casos, caso medio y peor de los casos. La complejidad espacial es  $O(n)$  en todos los casos, ya que se requiere espacio adicional para almacenar las sublistas y realizar la fusión. MergeSort es un algoritmo eficiente y estable en términos de ordenamiento, lo que lo hace adecuado para ordenar grandes conjuntos de datos.

## Mezcla Directa

El método de ordenamiento por mezcla directa, también conocido como MergeSort, es un algoritmo de ordenamiento eficiente basado en la técnica de "divide y vencerás". A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del algoritmo de mezcla directa (MergeSort).

Mejor de los casos:

En el mejor de los casos, el algoritmo de mezcla directa ya se encuentra en un estado ordenado o casi ordenado. En este caso, el algoritmo simplemente realiza las comparaciones necesarias y fusiona las sublistas sin la necesidad de realizar intercambios adicionales.

La complejidad temporal en el mejor de los casos para MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo divide repetidamente la lista en sublistas de tamaño medio hasta que cada sublista contenga un solo elemento, y luego fusiona las sublistas en orden creciente.

En cuanto a la complejidad espacial, MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(n)$ .



Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

División en sublistas: [1] [2] [3] [4] [5]

Fusión de sublistas: [1, 2, 3, 4, 5]

El algoritmo termina.

Caso medio:

En el caso medio, los elementos de la lista están desordenados de manera aleatoria. El algoritmo de mezcla directa divide repetidamente la lista en sublistas de tamaño medio y luego fusiona las sublistas en orden creciente.

La complejidad temporal en el caso medio para MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo divide repetidamente la lista en sublistas de tamaño medio hasta que cada sublista contenga un solo elemento, y luego fusiona las sublistas en orden creciente.

En cuanto a la complejidad espacial, MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el caso medio es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

División en sublistas: [5, 2] [4] [1, 3]

Fusión de sublistas: [2, 5] [4] [1, 3]

Fusión final: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo de mezcla directa debe realizar el máximo número de comparaciones y fusiones. Esto ocurre cuando la lista está en orden inverso.

La complejidad temporal en el peor de los casos para MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo divide repetidamente la lista en sublistas de tamaño medio hasta que cada sublista contenga un solo elemento, y luego fusiona las sublistas en orden creciente.

En cuanto a la complejidad espacial, MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el peor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

División en sublistas: [5] [4] [3] [2] [1]

Fusión de sublistas: [4, 5] [2, 3] [1]

Fusión final: [1, 2, 3, 4, 5]

El algoritmo termina.

En resumen, el algoritmo de mezcla directa (MergeSort) tiene una complejidad temporal de  $O(n \log n)$  en el mejor de los casos, caso medio y peor de los casos. La complejidad espacial es  $O(n)$  en todos los casos, ya que se requiere espacio adicional para almacenar las sublistas y realizar la fusión. MergeSort es un algoritmo eficiente y estable en términos de ordenamiento, lo que lo hace adecuado para ordenar grandes conjuntos de datos.

### Mezcla Natural

El método de ordenamiento por mezcla natural, también conocido como Natural MergeSort, es un algoritmo de ordenamiento eficiente que combina la técnica de mezcla directa con la detección de subsecuencias ya ordenadas en la lista. A continuación, te proporcionaré un análisis de los casos: mejor de los casos, caso medio y peor de los casos, junto con la complejidad en el tiempo y complejidad espacial del algoritmo de mezcla natural (Natural MergeSort).

Mejor de los casos:

En el mejor de los casos, el algoritmo de mezcla natural ya se encuentra en un estado ordenado. En este caso, el algoritmo simplemente detecta las subsecuencias ordenadas en la lista y las fusiona sin realizar intercambios adicionales.

La complejidad temporal en el mejor de los casos para Natural MergeSort es  $O(n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo solo necesita detectar las subsecuencias ordenadas y fusionarlas sin realizar operaciones adicionales.

En cuanto a la complejidad espacial, Natural MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el mejor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [1, 2, 3, 4, 5]

Pasos:

Detección de subsecuencias ordenadas: [1, 2, 3, 4, 5]

Fusión de subsecuencias: [1, 2, 3, 4, 5]



El algoritmo termina.

Caso medio:

En el caso medio, los elementos de la lista están desordenados de manera aleatoria. El algoritmo de mezcla natural detecta las subsecuencias ordenadas en la lista y las fusiona en orden creciente.

La complejidad temporal en el caso medio para Natural MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo necesita realizar operaciones de detección y fusión en cada subsecuencia ordenada.

En cuanto a la complejidad espacial, Natural MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el caso medio es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 2, 4, 1, 3]

Pasos:

Detección de subsecuencias ordenadas: [5], [2, 4], [1, 3]

Fusión de subsecuencias: [2, 4, 5], [1, 3]

Fusión final: [1, 2, 3, 4, 5]

El algoritmo termina.

Peor de los casos:

En el peor de los casos, el algoritmo de mezcla natural debe realizar el máximo número de comparaciones y fusiones. Esto ocurre cuando la lista está en orden inverso.

La complejidad temporal en el peor de los casos para Natural MergeSort es  $O(n \log n)$ , donde  $n$  es el número de elementos en la lista. Esto se debe a que el algoritmo necesita realizar operaciones de detección y fusión en cada subsecuencia ordenada.

En cuanto a la complejidad espacial, Natural MergeSort requiere espacio adicional para almacenar las sublistas y realizar la fusión. Por lo tanto, la complejidad espacial en el peor de los casos es  $O(n)$ .

Ejemplo:

Lista de entrada: [5, 4, 3, 2, 1]

Pasos:

Detección de subsecuencias ordenadas: [5], [4], [3], [2], [1]

Fusión de subsecuencias: [4, 5], [2, 3], [1]

Fusión final: [1, 2, 3, 4, 5]

El algoritmo termina.

En resumen, el algoritmo de mezcla natural (Natural MergeSort) tiene una complejidad temporal de  $O(n \log n)$  en el mejor de los casos, caso medio y peor de los casos. La complejidad espacial es  $O(n)$  en todos los casos, ya que se requiere espacio adicional para almacenar las sublistas y realizar la fusión. Natural MergeSort es un algoritmo eficiente y estable en términos de ordenamiento, especialmente cuando se encuentran subsecuencias ya ordenadas en la lista

- **Prueba de escritorio por cada algoritmo, usando los siguientes datos:**

23	-56	67	2	9	44	6	18	1	7
----	-----	----	---	---	----	---	----	---	---

### Algoritmos de ordenamiento internos

#### Burbuja Con Señal.


Código :

```
public class BurbujaConSeñal {  
    public static void bubbleSort(int[] array) {  
        int n = array.length;  
        boolean intercambio;  
  
        int señal = n - 1; // Señal inicialmente al final del arreglo  
  
        for (int i = 0; i < n - 1; i++) {  
            intercambio = false;  
  
            int nuevaSeñal = 0; // Nueva señal para cada iteración  
  
            for (int j = 0; j < señal; j++) {  
                if (array[j] > array[j + 1]) {  
                    // Intercambio de elementos  
  
                    int temp = array[j];
```

```
array[j] = array[j + 1];  
array[j + 1] = temp;  
intercambio = true;  
nuevaSeñal = j; // Actualización de la nueva señal  
}  
}  
  
señal = nuevaSeñal; // Actualizar la señal después de  
cada iteración  
  
if (!intercambio) {  
    // No se realizaron intercambios en esta iteración, el  
arreglo ya está ordenado  
    break;  
}  
}  
}
```

Prueba de escritorio:

Salida ×

 Arreglo: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Selecciona el tipo de modelo:




 Tipo

BurbujaConSeñal ▼

Aceptar Cancelar

Salida

 Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Aceptar

Doble Burbuja Con Señal.

Codigo:

```
public class DobleBurbujaConSeñal {  
    public static void doubleBubbleSort(int[] array) {  
        int n = array.length;  
        boolean intercambio;  
        int inicio = 0;  
        int fin = n - 1;  
        while (inicio < fin) {  
            intercambio = false;  
            // Recorrido hacia adelante (de inicio a fin)  
            for (int i = inicio; i < fin; i++) {  
                if (array[i] > array[i + 1]) {  
                    // Intercambio de elementos  
                    int temp = array[i];  
                    array[i] = array[i + 1];  
                    array[i + 1] = temp;  
                }  
            }  
            fin = fin - 1;  
        }  
    }  
}
```

```
intercambio = true;

}

}

if (!intercambio) {

    // No se realizaron intercambios en el recorrido hacia
    // adelante, el arreglo ya está ordenado

    break;

}

intercambio = false;

fin--;

// Recorrido hacia atrás (de fin a inicio)

for (int i = fin - 1; i >= inicio; i--) {

    if (array[i] > array[i + 1]) {

        // Intercambio de elementos

        int temp = array[i];

        array[i] = array[i + 1];

        array[i + 1] = temp;

        intercambio = true;

    }

}

if (!intercambio) {
```

```
// No se realizaron intercambios en el recorrido hacia  
atrás, el arreglo ya está ordenado
```

```
break;
```

```
}
```

```
inicio++;
```

```
}
```

```
}
```

Prueba de escritorio:

Salida

?

Arreglo restaurado: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Aceptar

Selecciona el tipo de modelo:

Tipo

DobleBurbujaCon Señal

Aceptar Cancelar

Salida

?

Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Aceptar

Shell (incrementos - decrementos)

```
public class ShellSort {  
    public static void shellSort(int[] array) {  
        int n = array.length;  
        int brecha = 1;  
        // Calcular la secuencia de incrementos y decrementos  
        while (brecha < n / 3) {
```

```
brecha = brecha * 3 + 1;

}

while (brecha >= 1) {

    // Aplicar el algoritmo de inserción para cada subarreglo  
con la brecha actual

    for (int i = brecha; i < n; i++) {

        int valorActual = array[i];

        int j = i;

        // Desplazar los elementos hacia la derecha hasta  
encontrar la posición correcta

        while (j >= brecha && array[j - brecha] > valorActual) {

            array[j] = array[j - brecha];

            j -= brecha;

        }

        array[j] = valorActual;

    }

    brecha = brecha / 3;

}

}
```

Prueba de escritorio:

Salida

?

Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Aceptar

Selecciona el tipo de modelo:

?

Tipo

Shell(incrementos-decrementos)

Aceptar Cancelar

Salida

?

Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Aceptar

[Selección directa](#) [Inserción directa](#)

Código:

```
public class SeleccionDirectaInsercion {  
  
    public static void seleccionDirectaInsercionSort(int[]  
array) {  
  
        int n = array.length;  
  
        for (int i = 0; i < n - 1; i++) {  
  
            int indiceMinimo = i;  
  
            // Buscar el índice del elemento más pequeño en el  
subarreglo no ordenado  
  
            for (int j = i + 1; j < n; j++) {  
  
                if (array[j] < array[indiceMinimo]) {  
  
                    indiceMinimo = j;  
  
                }  
  
            }  
  
        }  
  
    }  
}
```



```
// Intercambiar el elemento más pequeño con el primer
// elemento del subarreglo no ordenado

int temp = array[indiceMinimo];

array[indiceMinimo] = array[i];

array[i] = temp;

// Insertar el elemento en su posición correcta en el
// subarreglo ordenado

for (int k = i + 1; k > 0 && array[k] < array[k - 1]; k--) {

    temp = array[k];

    array[k] = array[k - 1];

    array[k - 1] = temp;

}

}

}}

public class SeleccionDirecta {

    public static void seleccionDirectaSort(int[] array) {

        int n = array.length;

        for (int i = 0; i < n - 1; i++) {

            int indiceMinimo = i;


            // Buscar el índice del elemento más pequeño en el
            // subarreglo no ordenado

            for (int j = i + 1; j < n; j++) {
```


```
if (array[j] < array[indiceMinimo]) {  
    indiceMinimo = j;  
}  
}  
  
// Intercambiar el elemento más pequeño con el primer  
// elemento del subarreglo no ordenado  
  
int temp = array[indiceMinimo];  
array[indiceMinimo] = array[i];  
array[i] = temp;  
}  
}}
```

Prueba de escritorio:


Salida ×

 Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Selecciona el tipo de modelo: ×

 Tipo

Salida ×

 Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Binaria

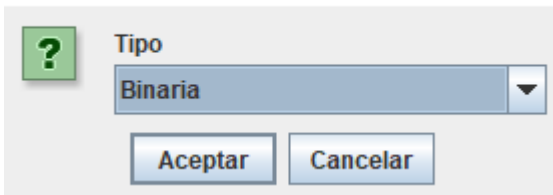
Codigo:

```
public class BusquedaBinaria {  
  
    public static int busquedaBinaria(int[] array, int  
    elemento) {  
  
        int izquierda = 0;  
  
        int derecha = array.length - 1;  
  
        while (izquierda <= derecha) {  
  
            int medio = izquierda + (derecha - izquierda) / 2;  
  
            // Si el elemento está en el medio, se retorna su  
            posición  
  
            if (array[medio] == elemento) {  
  
                return medio;  
  
            }  
  
            // Si el elemento es mayor, se descarta la mitad  
            izquierda del arreglo  
  
            if (array[medio] < elemento) {  
  
                izquierda = medio + 1;  
  
            }  
  
            // Si el elemento es menor, se descarta la mitad derecha  
            del arreglo  
  
            else {  
  
                derecha = medio - 1;  
  
            }  
  
        }  
  
    }  
}
```

```
}  
  
// Si el elemento no se encuentra en el arreglo, se  
// retorna -1  
return -1;  
}}
```

Prueba de escritorio:

Selecciona el tipo de modelo: X



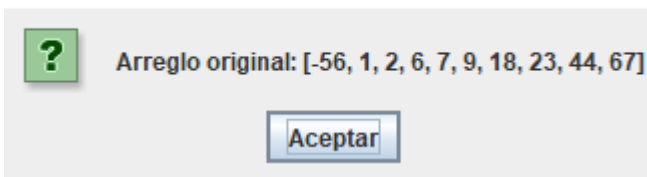
?

Tipo

Binaria

Aceptar Cancelar

Salida X



?

Arreglo original: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Aceptar

HeapSort


Codigo:

```
public class HeapSort {  
    public static void heapSort(int[] array) {  
        int n = array.length;  
        // Construir el montículo (heap)  
        for (int i = n / 2 - 1; i >= 0; i--) {  
            heapify(array, n, i);  
        }  
        // Extraer el elemento máximo repetidamente y reconstruir  
        // el montículo
```

```
for (int i = n - 1; i > 0; i--) {  
    // Mover el elemento máximo al final del arreglo  
    int temp = array[0];  
    array[0] = array[i];  
    array[i] = temp;  
    // Reconstruir el montículo en la porción no ordenada del  
    arreglo  
    heapify(array, i, 0);  
}
```

Prueba de escritorio:


Selecciona el tipo de modelo: X

 Tipo  
HeapSort ▼  
Aceptar Cancelar

Salida X

 Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]  
Aceptar

Salida X

 Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]  
Aceptar

[Quicksoft recursivo](#)


Código:

```
public class QuickSortRecursivo {  
    public static void quickSort(int[] array, int inicio, int  
    fin) {  
        if (inicio < fin) {  
            // Particionar el arreglo y obtener la posición del  
            pivote
```


```
int indicePivote = particion(array, inicio, fin);  
  
// Recursivamente ordenar los subarreglos izquierdo y  
derecho del pivote  
  
quickSort(array, inicio, indicePivote - 1);  
quickSort(array, indicePivote + 1, fin);  
  
}  
  
}
```

Prueba de escritorio:


Selecciona el tipo de modelo: X

 Tipo  
QuicksoftRecursivo

Salida X

 Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Salida X

 Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

## Algoritmos de ordenamiento externos

### Intercalación

```
public class OrdenamientoIntercalacion {  
    public static void mergeSort(int[] array) {  
        if (array.length <= 1) {  
            return; // Caso base: el arreglo está ordenado  
        }  
    }  
}
```

```
int medio = array.length / 2;

// Dividir el arreglo en dos subarreglos

int[] subarregloIzquierdo = Arrays.copyOfRange(array, 0,
medio);

int[] subarregloDerecho = Arrays.copyOfRange(array,
medio, array.length);

// Ordenar recursivamente los subarreglos

mergeSort(subarregloIzquierdo);

mergeSort(subarregloDerecho);

// Combinar los subarreglos ordenados

merge(subarregloIzquierdo, subarregloDerecho, array);
}

private static void merge(int[] subarregloIzquierdo,
int[] subarregloDerecho, int[] array) {

int i = 0; // Índice para recorrer el subarreglo
izquierdo

int j = 0; // Índice para recorrer el subarreglo derecho

int k = 0; // Índice para recorrer el arreglo final

// Combinar los elementos de los subarreglos en el
arreglo final en orden ascendente

while (i < subarregloIzquierdo.length && j <
subarregloDerecho.length) {

if (subarregloIzquierdo[i] <= subarregloDerecho[j]) {


array[k] = subarregloIzquierdo[i];
```



```
i++;  
  
} else {  
  
array[k] = subarregloDerecho[j];  
  
j++;  
  
}  
  
k++;  
  
}  
  
// Copiar los elementos restantes del subarreglo  
izquierdo, si los hay  
  
while (i < subarregloIzquierdo.length) {  
  
array[k] = subarregloIzquierdo[i];  
  
i++;  
  
k++;  
  
}  
  
// Copiar los elementos restantes del subarreglo derecho,  
si los hay  
  
while (j < subarregloDerecho.length) {  
  
array[k] = subarregloDerecho[j];  
  
j++;  
  
k++;  
  
}
```

Prueba de escritorio:


Selecciona el tipo de modelo: ✕

 Tipo

Intercalación ▼


Aceptar Cancelar

Salida ✕

 Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Aceptar

Salida ✕

 Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Aceptar

## Mezcla Directa

Código:

```
public class MezclaDirecta {  
    public static void mergeSort(int[] array) {  
        if (array.length <= 1) {  
            return; // El arreglo ya está ordenado o es vacío  
        }  
        int medio = array.length / 2;  
        int[] mitadIzquierda = Arrays.copyOfRange(array, 0,  
            medio);  
        int[] mitadDerecha = Arrays.copyOfRange(array, medio,  
            array.length);  
        mergeSort(mitadIzquierda);
```

```
mergeSort(mitadDerecha);

merge(array, mitadIzquierda, mitadDerecha);
}

private static void merge(int[] array, int[]
mitadIzquierda, int[] mitadDerecha) {

    int i = 0; // Índice para recorrer la mitad izquierda del
    arreglo

    int j = 0; // Índice para recorrer la mitad derecha del
    arreglo

    int k = 0; // Índice para recorrer el arreglo original

    while (i < mitadIzquierda.length && j <
    mitadDerecha.length) {

        if (mitadIzquierda[i] <= mitadDerecha[j]) {

            array[k] = mitadIzquierda[i];

            i++;

        } else {

            array[k] = mitadDerecha[j];

            j++;

        }

        k++;

    }


    while (i < mitadIzquierda.length) {

        array[k] = mitadIzquierda[i];
```


```
i++;  
k++;  
}  
while (j < mitadDerecha.length) {  
array[k] = mitadDerecha[j];  
j++;  
k++;  
}
```

Prueba de escritorio:


Selecciona el tipo de modelo: X

 Tipo  
Mezcla Directa ▼  
Aceptar Cancelar

Salida X

 Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]  
Aceptar

Salida X

 Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]  
Aceptar

## Mezcla Natural

Código:

```
public class MezclaNatural {  
public static void naturalMergeSort(int[] array) {  
int n = array.length;  
int izquierda = 0;
```

```
int derecha = n - 1;

while (izquierda < derecha) {

int min = encontrarMin(array, izquierda, derecha);

if (min == derecha) {

derecha--;

} else {

mezclar(array, izquierda, min, derecha);

derecha--;

}

if (izquierda == derecha) {

izquierda++;


derecha = n - 1;

}

}
```

Prueba de escritorio:

Selecciona el tipo de modelo: ×

 Tipo

Salida



Arreglo original: [23, -56, 67, 2, 9, 44, 6, 18, 1, 7]

Aceptar

Salida



Arreglo ordenado: [-56, 1, 2, 6, 7, 9, 18, 23, 44, 67]

Aceptar