TECNOLOGICO NACIONAL DE MEXICO

INSTITUTO TECNOLOGICO DE ORIZABA

Asignatura: Estructura de Datos

Carrera: Ingeniería en Informática

Estudiantes:

- Raúl García Jiménez - No. Control 19010405
- Dorantes Rodríguez Diego Yael - No. Control 21010184
- Carlos Eduardo Nicanor Flores- No. Control 21010205
- Axel Reyes Guevara-

Profesora: María Jacinta Martínez Castillo

Grupo: 3a3A

Fecha de entrega: 30/05/2023

Algoritmo de ordenamiento internos

- **BURBUJA CON SEÑAL**

El algoritmo de ordenación burbuja es un algoritmo simple que ordena una lista de elementos comparando pares adyacentes y haciendo intercambios si están en el orden incorrecto. Se puede implementar el algoritmo de burbuja con una señal para indicar cuándo la lista está completamente ordenada.

Ejemplo de implementación en pseudocódigo:

```
procedure BurbujaConSenal(lista)
    n := longitud(lista)
    intercambiado := verdadero
    fin := n - 1

    while intercambiado hacer
        intercambiado := falso
        for i := 0 to fin - 1 hacer
            si lista[i] > lista[i+1] entonces
                intercambiar(lista[i], lista[i+1])
                intercambiado := verdadero
            fin := fin - 1

    retornar lista
```

En este algoritmo, utilizamos una variable “intercambiado” para indicar si se realizó algún intercambio durante un recorrido completo de la lista. Si no se realiza ningún intercambio en un recorrido completo, significa que la lista está ordenada y podemos detener el algoritmo.

Además, utilizamos la variable “fin” para evitar comparaciones innecesarias. Después de cada recorrido completo, el elemento más grande se coloca en la

posición correcta, por lo que podemos reducir la cantidad de elementos a considerar en el siguiente recorrido.

= ANÁLISIS DE EFICIENCIA =

En términos de eficiencia, el algoritmo de burbuja no es muy eficiente, especialmente para listas grandes, ya que su complejidad cuadrática puede resultar en un tiempo de ejecución largo. Hay otros algoritmos de ordenación más eficientes, como el algoritmo de ordenación rápida (quicksort) o el algoritmo de ordenación por mezcla (merge sort), que tienen complejidad O(n log n) en el peor caso y en promedio, y son más eficientes para grandes conjuntos de datos. Sin embargo, el algoritmo de burbuja puede ser útil para listas pequeñas o en casos donde la lista ya está casi ordenada, ya que en esos casos su rendimiento puede ser aceptable.

= ANÁLISIS DE LOS CASOS =

Veamos algunos ejemplos para analizar el rendimiento del algoritmo de burbuja en diferentes casos:

1. Peor caso:

Supongamos que tenemos una lista desordenada con 5 elementos: [5, 4, 3, 2, 1].

El algoritmo de burbuja realizará los siguientes pasos:

- Comparación y intercambio: (5, 4), (5, 3), (5, 2), (5, 1) -> [4, 3, 2, 1, 5]
- Comparación y intercambio: (4, 3), (4, 2), (4, 1) -> [3, 2, 1, 4, 5]
- Comparación y intercambio: (3, 2), (3, 1) -> [2, 1, 3, 4, 5]

- Comparación y intercambio: (2, 1) -> [1, 2, 3, 4, 5]

En este caso, se realizan un total de 10 comparaciones e intercambios para ordenar la lista de 5 elementos. La cantidad de operaciones aumenta cuadráticamente con el tamaño de la lista.

2. Mejor caso:

Supongamos que tenemos una lista ordenada con 5 elementos: [1, 2, 3, 4, 5].

El algoritmo de burbuja realizará los siguientes pasos:

- Comparación: (1, 2), (2, 3), (3, 4), (4, 5)

En este caso, no se realiza ningún intercambio ya que la lista ya está ordenada. Se realizaron un total de 4 comparaciones para confirmar que la lista está ordenada. La cantidad de operaciones es lineal con el tamaño de la lista.

3. Caso promedio:

Supongamos que tenemos una lista desordenada con 5 elementos: [3, 1, 4, 2, 5].

El algoritmo de burbuja realizará los siguientes pasos:

- Comparación y intercambio: (3, 1), (3, 4), (4, 2), (4, 5) -> [1, 3, 2, 4, 5]
- Comparación y intercambio: (1, 3), (3, 2), (3, 4) -> [1, 2, 3, 4, 5]
- Comparación: (1, 2), (2, 3), (3, 4), (4, 5)

En este caso, se realizan un total de 8 comparaciones e intercambios para ordenar la lista de 5 elementos. La cantidad de operaciones es cercana a la cantidad en el peor caso, pero puede variar dependiendo de la distribución de los elementos en la lista.

= COMPLEJIDAD EN EL TIEMPO Y COMPLEJIDAD ESPACIAL =

Complejidad en tiempo:

La complejidad temporal del algoritmo de burbuja es O(n^2), donde n es la longitud de la lista a ordenar. Esto se debe a que el algoritmo realiza comparaciones y posibles intercambios para cada par de elementos en la lista. En el peor caso, se deben realizar n*(n-1)/2 comparaciones, lo cual se aproxima a n^2/2 comparaciones. Por lo tanto, la complejidad en tiempo del algoritmo de burbuja es cuadrática.

Complejidad espacial:

La complejidad espacial del algoritmo de burbuja es O(1), lo que significa que no requiere memoria adicional en función del tamaño de la lista. El algoritmo opera directamente en la lista de entrada y no utiliza estructuras de datos auxiliares significativas. Solo se necesitan unas pocas variables adicionales para realizar los intercambios y las comparaciones. Por lo tanto, la complejidad espacial es constante y no depende del tamaño de la lista.

- **ALGORITMO DOBLE BURBUJA CON SEÑAL**

El algoritmo de doble burbuja con señal es una variante del algoritmo de burbuja que realiza pasadas de izquierda a derecha y de derecha a izquierda en cada iteración, intercambiando los elementos adyacentes si están en el orden incorrecto. La señal se utiliza para indicar cuándo se ha realizado algún intercambio en una pasada, de manera similar al algoritmo de burbuja con señal.

Ejemplo de implementación en pseudocódigo:

```
procedure DobleBurbujaConSenal(lista)
    n := longitud(lista)
    intercambiado := verdadero

    mientras intercambiado hacer
        intercambiado := falso

        para i := 0 hasta n-2 hacer
            si lista[i] > lista[i+1] entonces
                intercambiar(lista[i], lista[i+1])
                intercambiado := verdadero

        si no intercambiado entonces
            salir del bucle

        intercambiado := falso

        para i := n-2 hasta 0 hacer
            si lista[i] > lista[i+1] entonces
                intercambiar(lista[i], lista[i+1])
                intercambiado := verdadero

    retornar lista
```

En este algoritmo, se realizan dos pasadas en cada iteración: una de izquierda a derecha y otra de derecha a izquierda. En cada pasada, se compara y se intercambian los elementos adyacentes si están en el orden incorrecto. Después de cada par de pasadas, se verifica si se ha realizado algún intercambio. Si no se realiza ningún intercambio, significa que la lista está completamente ordenada y se puede salir del bucle.

Este algoritmo tiene una complejidad temporal similar al algoritmo de burbuja, es decir, O(n^2) en el peor caso. Sin embargo, debido a las pasadas de derecha a izquierda, el algoritmo puede realizar menos comparaciones e intercambios en

algunos casos, lo que puede mejorar su rendimiento en comparación con el algoritmo de burbuja tradicional en ciertas situaciones.

= ANALISIS DE EFICIENCIA =

En términos de eficiencia, el algoritmo de doble burbuja con señal no mejora significativamente la complejidad temporal del algoritmo de burbuja. Aunque puede realizar menos comparaciones e intercambios en algunos casos, sigue siendo un algoritmo cuadrático y no es tan eficiente como otros algoritmos de ordenación más avanzados, como quicksort o mergesort, que tienen complejidad O(n log n) en el peor caso y en promedio. Por lo tanto, el algoritmo de doble burbuja con señal puede ser adecuado para listas pequeñas o en casos donde la lista ya está casi ordenada, pero no es recomendable para grandes conjuntos de datos.

= ANALISIS DE LOS CASOS =

Vamos a analizar el rendimiento del algoritmo de doble burbuja con señal en diferentes casos utilizando ejemplos:

1. Peor caso:

Supongamos que tenemos una lista desordenada con 5 elementos: [5, 4, 3, 2, 1].

El algoritmo de doble burbuja con señal realizará los siguientes pasos:

- Pasada de izquierda a derecha: comparaciones e intercambios: (5, 4), (4, 3), (3, 2), (2, 1) -> [4, 3, 2, 1, 5]

- Pasada de derecha a izquierda: comparaciones e intercambios: (5, 1), (4, 2), (3, 1) -> [4, 2, 1, 3, 5]

- Pasada de izquierda a derecha: comparaciones e intercambios: (4, 2), (2, 1) -> [2, 1, 4, 3, 5]

- Pasada de derecha a izquierda: comparaciones e intercambios: (4, 1), (3, 1) -> [1, 2, 3, 4, 5]

En este caso, se realizaron un total de 14 comparaciones e intercambios para ordenar la lista de 5 elementos. La cantidad de operaciones aumenta cuadráticamente con el tamaño de la lista.

2. Mejor caso:

Supongamos que tenemos una lista ordenada con 5 elementos: [1, 2, 3, 4, 5].

El algoritmo de doble burbuja con señal realizará los siguientes pasos:

- Pasada de izquierda a derecha: no se realiza ningún intercambio ya que los elementos están en orden.

- Pasada de derecha a izquierda: no se realiza ningún intercambio.

En este caso, no se realiza ningún intercambio en ninguna de las pasadas ya que la lista ya está ordenada. Se realizan un total de 2 comparaciones para confirmar que la lista está ordenada.

3. Caso promedio:

Supongamos que tenemos una lista desordenada con 5 elementos: [3, 1, 4, 2, 5].

El algoritmo de doble burbuja con señal realizará los siguientes pasos:

- Pasada de izquierda a derecha: comparaciones e intercambios: (3, 1), (4, 2) -> [1, 3, 2, 4, 5]

- Pasada de derecha a izquierda: comparaciones e intercambios: (4, 2), (5, 4) -> [1, 3, 2, 4, 5]

- Pasada de izquierda a derecha: comparaciones e intercambios: (3, 2), (4, 2) -> [1, 2, 3, 4, 5]

- Pasada de derecha a izquierda: no se realiza ningún intercambio.

En este caso, se realizaron un total de 6 comparaciones e intercambios para ordenar la lista de 5 elementos. La cantidad de operaciones es menor que en el peor caso, debido a las pasadas de derecha a izquierda que permiten corregir el orden de algunos elementos de manera más eficiente.

Estos ejemplos ilustran cómo el rendimiento del algoritmo de doble burbuja con señal puede variar según la disposición de los elementos en la lista. Al igual que el algoritmo de burbuja, el peor caso ocurre cuando la lista está completamente desordenada, el mejor caso ocurre cuando la lista ya está ordenada, y el caso promedio depende de la distribución de los elementos. Sin embargo, el algoritmo de doble burbuja con señal puede realizar menos comparaciones e intercambios en comparación con el algoritmo de burbuja tradicional debido a las pasadas de derecha a izquierda, lo que puede mejorar su rendimiento en algunos casos.

= COMPLEJIDAD EN EL TIEMPO Y COMPLEJIDAD ESPACIAL =

Complejidad en tiempo:

La complejidad temporal del algoritmo de doble burbuja con señal es O(n^2), donde n es la longitud de la lista a ordenar. Esto se debe a que se realizan pasadas de izquierda a derecha y de derecha a izquierda, y en cada pasada se comparan y posiblemente se intercambian los elementos adyacentes. En el peor caso, se deben realizar n*(n-1)/2 comparaciones e intercambios en total, lo cual se aproxima a n^2/2. Por lo tanto, la complejidad en tiempo del algoritmo de doble burbuja con señal es cuadrática.

Complejidad espacial:

La complejidad espacial del algoritmo de doble burbuja con señal es O(1), al igual que el algoritmo de burbuja con señal. No se requiere memoria adicional en función del tamaño de la lista, ya que el algoritmo opera directamente en la lista de entrada y utiliza solo un número constante de variables adicionales. Por lo tanto, la complejidad espacial es constante y no depende del tamaño de la lista.

- **SHELL (INCREMENTOS - DECREMENTOS)**

El algoritmo de Shell, también conocido como Shell Sort, es un algoritmo de ordenación que utiliza un incremento y decremento progresivo para realizar comparaciones y mover elementos en una lista. A medida que avanza el algoritmo, el incremento se reduce gradualmente hasta llegar a 1, lo que finaliza el proceso de ordenación.

Ejemplo de implementación en pseudocódigo utilizando el incremento y decremento progresivo:

```
procedure ShellSort(lista)
    n := longitud(lista)
    incremento := n / 2

    mientras incremento > 0 hacer
        para i := incremento hasta n hacer
            temp := lista[i]
            j := i

            mientras j >= incremento y lista[j - incremento] > temp hacer
                lista[j] := lista[j - incremento]
                j := j - incremento

            lista[j] := temp

        incremento := incremento / 2

    retornar lista
```

En este algoritmo, el incremento inicial se establece como la mitad de la longitud de la lista (n/2). Luego, el algoritmo realiza un bucle mientras el incremento sea mayor que 0. Dentro del bucle, se realiza un bucle que recorre la lista desde el incremento hasta el final. Se compara cada elemento con el elemento ubicado a una distancia de incremento, y si es necesario, se realiza el intercambio. El proceso se repite para

diferentes valores de j, donde j = i - incremento. El incremento se reduce a la mitad en cada iteración del bucle externo hasta que se alcanza un incremento de 1.

El algoritmo de Shell utiliza el concepto de incremento para reducir la cantidad de elementos desordenados a considerar en cada iteración, lo que puede mejorar el rendimiento en comparación con algoritmos de ordenación más simples. Sin embargo, la eficiencia del algoritmo de Shell puede variar según la secuencia de incrementos utilizada.

= ANALISIS DE EFICIENCIA =

El análisis de eficiencia del algoritmo Shell Sort con incremento y decremento progresivo puede ser un poco más complejo debido a la variedad de secuencias de incrementos que se pueden utilizar. Sin embargo, en términos generales, el rendimiento del algoritmo depende en gran medida de la secuencia de incrementos utilizada.

El algoritmo de Shell Sort con incremento y decremento progresivo puede ofrecer un rendimiento eficiente, especialmente si se utiliza una secuencia de incrementos bien elegida. Aunque su complejidad en tiempo puede variar según la secuencia de incrementos utilizada, en general, el algoritmo tiene una complejidad en tiempo promedio mejor que O(n^2). Sin embargo, para garantizar un rendimiento óptimo, es importante elegir una secuencia de incrementos adecuada.

= ANALISIS DE LOS CASOS =

Vamos a analizar el rendimiento del algoritmo de Shell Sort en diferentes casos utilizando ejemplos:

1. Peor caso:

Supongamos que tenemos una lista inversamente ordenada con 6 elementos: [6, 5, 4, 3, 2, 1].

Si utilizamos la secuencia de incremento estática de Knuth (1, 4, 13, 40, ...), el algoritmo de Shell Sort realizará los siguientes pasos:

- Primera pasada con incremento 4: [2, 1, 4, 3, 6, 5]
- Segunda pasada con incremento 1: [1, 2, 3, 4, 5, 6]

En este caso, el algoritmo realiza un total de 9 comparaciones e intercambios para ordenar la lista de 6 elementos. La cantidad de operaciones aumenta cuadráticamente con el tamaño de la lista debido a la secuencia de incremento estática.

2. Mejor caso:

Supongamos que tenemos una lista ordenada con 6 elementos: [1, 2, 3, 4, 5, 6].

Si utilizamos la secuencia de incremento estática de Knuth (1, 4, 13, 40, ...), el algoritmo de Shell Sort realizará los siguientes pasos:

- Primera pasada con incremento 4: no se realiza ningún intercambio ya que los elementos están en orden.
- Segunda pasada con incremento 1: no se realiza ningún intercambio.

En este caso, no se realiza ningún intercambio en ninguna de las pasadas ya que la lista ya está ordenada. Se realizan un total de 5 comparaciones para confirmar que la lista está ordenada.

3. Caso promedio:

Supongamos que tenemos una lista parcialmente ordenada con 6 elementos: [1, 4, 2, 5, 3, 6].

Si utilizamos la secuencia de incremento estática de Knuth (1, 4, 13, 40, ...), el algoritmo de Shell Sort realizará los siguientes pasos:

- Primera pasada con incremento 4: [1, 3, 2, 5, 4, 6]
- Segunda pasada con incremento 1: [1, 2, 3, 4, 5, 6]

En este caso, se realizaron un total de 8 comparaciones e intercambios para ordenar la lista de 6 elementos. La cantidad de operaciones es menor que en el peor caso debido a la parcial ordenación de la lista.

Estos ejemplos ilustran cómo el rendimiento del algoritmo de Shell Sort puede variar según la secuencia de incremento utilizada y la distribución de los elementos en la lista. En el peor caso, el rendimiento puede ser similar a los algoritmos cuadráticos, mientras que en el mejor caso y en casos promedio con una distribución parcialmente ordenada, el rendimiento puede ser mejor, aproximándose a una complejidad O(n log n) o incluso mejor. La selección de una secuencia de incremento adecuada es clave para optimizar el rendimiento del algoritmo de Shell Sort.

= COMPLEJIDAD EN EL TIEMPO Y COMPLEJIDAD ESPACIAL =

Complejidad en el tiempo:

La complejidad temporal del algoritmo de Shell Sort es difícil de determinar exactamente debido a la variedad de secuencias de incremento posibles. En el peor caso, con una secuencia de incremento estática no óptima, como la secuencia de Knuth, la complejidad temporal es O(n^2), similar a los algoritmos de ordenación cuadráticos. Sin embargo, al utilizar secuencias de incremento más eficientes, como

la secuencia de Sedgewick, el algoritmo puede tener un rendimiento mucho mejor, con una complejidad promedio estimada en O(n log n) o incluso mejor en algunos casos.

Complejidad espacial:

La complejidad espacial del algoritmo de Shell Sort es O(1), lo que significa que no requiere memoria adicional en función del tamaño de la lista. El algoritmo opera directamente en la lista de entrada y utiliza solo un número constante de variables auxiliares. Por lo tanto, la complejidad espacial es constante y no depende del tamaño de la lista.

Selección Directa

Análisis de Eficiencia

La selección directa (selection sort en inglés) es un algoritmo de ordenamiento simple que funciona encontrando repetidamente el elemento mínimo de una lista y colocándolo al principio. A continuación, se busca el siguiente elemento mínimo de la lista restante y se coloca después del elemento previamente ordenado. Este proceso se repite hasta que toda la lista esté ordenada.

Para analizar la eficiencia del algoritmo de selección directa, podemos utilizar la notación de O grande (Big O notation). La notación O grande nos permite expresar la complejidad de un algoritmo en términos de su crecimiento relativo en función del tamaño de la entrada.

El análisis de eficiencia del algoritmo de selección directa es el siguiente:

Mejor caso: O(n^2)

En el mejor caso, la lista ya está ordenada o casi ordenada. Sin embargo, el algoritmo de selección directa todavía realizará comparaciones para encontrar el elemento mínimo en cada iteración, lo que resultará en una complejidad cuadrática.

En cada iteración, se realizan n-1 comparaciones en el peor caso, donde n es el tamaño de la lista.

Caso promedio: O(n^2)

En promedio, el algoritmo de selección directa realiza (n^2)/2 comparaciones y (n^2)/2 intercambios. Esto se debe a que, en cada iteración, se encuentra el mínimo en la lista restante y se realiza un intercambio con el elemento actual. La complejidad cuadrática se mantiene en el caso promedio.

Peor caso: O(n^2)

En el peor caso, el algoritmo de selección directa realiza (n^2)/2 comparaciones y (n^2)/2 intercambios, donde n es el tamaño de la lista. En cada iteración, se encuentra el mínimo en la lista restante y se realiza un intercambio con el elemento actual. La complejidad cuadrática se alcanza cuando la lista está completamente desordenada.

En términos de eficiencia, el algoritmo de selección directa tiene una complejidad cuadrática de O(n^2) en todos los casos (mejor, promedio y peor). Esto significa que el tiempo de ejecución del algoritmo aumenta cuadráticamente a medida que el tamaño de la lista aumenta.

Es importante tener en cuenta que, aunque el análisis de eficiencia se basa en estimaciones teóricas, la selección directa generalmente se considera menos eficiente que otros algoritmos de ordenamiento más rápidos, como el algoritmo de ordenamiento rápido (quicksort) o el algoritmo de ordenamiento por mezcla (merge sort), que tienen una complejidad promedio de O(n log n).

Análisis de los Casos: mejor de los casos, caso medio y peor de los casos

El análisis de casos para el algoritmo de selección directa implica considerar diferentes escenarios en términos de la disposición de los elementos en la lista. Aquí están los casos clave a tener en cuenta:

Mejor caso: La lista ya está completamente ordenada. En el mejor caso, no se realizarán intercambios en absoluto, ya que los elementos ya están en su posición correcta. Sin embargo, el algoritmo seguirá realizando comparaciones para encontrar el elemento mínimo en cada iteración. La complejidad de tiempo sería O(n^2) ya que aún se deben realizar comparaciones para encontrar el mínimo en cada iteración.

Caso promedio: Los elementos están en una disposición aleatoria en la lista. En promedio, el algoritmo de selección directa realiza (n^2)/2 comparaciones y (n^2)/2 intercambios, donde n es el tamaño de la lista. Esto se debe a que, en cada iteración, se encuentra el elemento mínimo en la lista restante y se realiza un intercambio con el elemento actual. La complejidad de tiempo sería O(n^2).

Peor caso: La lista está completamente desordenada. En el peor caso, el algoritmo de selección directa realizará la mayor cantidad de comparaciones e intercambios posibles. En cada iteración, se encuentra el elemento mínimo en la lista restante y se realiza un intercambio con el elemento actual. La complejidad de tiempo sería O(n^2) ya que se realizan (n^2)/2 comparaciones y (n^2)/2 intercambios.

Complejidad en el Tiempo

La complejidad en el tiempo del algoritmo de selección directa es O(n^2), donde n es el tamaño de la lista que se está ordenando. Esto implica que el tiempo de ejecución del algoritmo crece cuadráticamente a medida que aumenta el tamaño de la lista.

El algoritmo de selección directa tiene un enfoque simple pero ineficiente para ordenar una lista. En cada iteración, busca el elemento mínimo en la lista restante y lo coloca en su posición correcta, repitiendo este proceso hasta que toda la lista esté ordenada.

En el peor caso, el algoritmo de selección directa realiza aproximadamente (n^2)/2 comparaciones y (n^2)/2 intercambios, lo que da lugar a una complejidad cuadrática de O(n^2). Esto significa que el número de operaciones necesarias para ordenar la lista aumenta proporcionalmente al cuadrado del tamaño de la lista.

Aunque en el mejor caso y en casos promedio también se alcanza una complejidad de O(n^2), en estos escenarios, el número de comparaciones y asignaciones puede ser menor en comparación con el peor caso. Sin embargo, el crecimiento general sigue siendo cuadrático.

Complejidad Espacial

La complejidad espacial del algoritmo de selección directa es O(1), es decir, constante. Esto significa que el algoritmo no requiere memoria adicional proporcional al tamaño de la lista que se está ordenando.

El algoritmo de selección directa opera directamente en la lista original, realizando comparaciones y permutaciones entre los elementos existentes. No se requiere crear estructuras de datos adicionales ni asignar memoria adicional para almacenar el resultado del proceso de ordenamiento.

El espacio adicional utilizado por el algoritmo de selección directa se limita a un número constante de variables auxiliares utilizadas para realizar las comparaciones y los intercambios entre los elementos de la lista. Estas variables ocupan una cantidad fija de memoria independientemente del tamaño de la lista.

Por lo tanto, la complejidad espacial del algoritmo de selección directa es O(1), lo que implica que el consumo de memoria no aumenta a medida que aumenta el tamaño de la lista. Esto hace que el algoritmo de selección directa sea eficiente en términos de espacio.

Inserción Directa

Análisis de Eficiencia

El algoritmo de inserción directa es eficiente para listas pequeñas o listas que ya están parcialmente ordenadas. Sin embargo, a medida que el tamaño de la lista

aumenta, el número de comparaciones y movimientos de elementos también aumenta cuadráticamente, lo que puede hacer que el algoritmo sea menos eficiente en comparación con otros algoritmos de ordenamiento más rápidos, como quicksort o mergesort, que tienen una complejidad promedio de O(n log n).

Análisis de los Casos: mejor de los casos, caso medio y peor de los casos

Mejor caso:

La lista ya está completamente ordenada. En el mejor caso, el algoritmo de inserción directa realiza n-1 comparaciones, donde n es el tamaño de la lista. No se requieren movimientos de elementos, ya que cada elemento ya está en su posición correcta. La complejidad de tiempo en el mejor caso es O(n).

Caso promedio:

Los elementos están en una disposición aleatoria en la lista. En promedio, el algoritmo de inserción directa realizará aproximadamente n^2/4 comparaciones y n^2/4 movimientos de elementos. La cantidad de comparaciones y movimientos aumenta cuadráticamente a medida que crece el tamaño de la lista. La complejidad de tiempo en el caso promedio es O(n^2).

Peor caso:

La lista está completamente desordenada en orden descendente. En el peor caso, el algoritmo de inserción directa realizará aproximadamente n^2/2 comparaciones y n^2/2 movimientos de elementos. Cada elemento debe ser comparado y desplazado hasta su posición correcta al principio de la lista. La complejidad de tiempo en el peor caso es O(n^2).

Complejidad en el Tiempo

La complejidad en el tiempo del algoritmo de inserción directa es O(n^2) en todos los casos (mejor, promedio y peor). Esto implica que el tiempo de ejecución del algoritmo aumenta cuadráticamente a medida que aumenta el tamaño de la lista.

Complejidad Espacial

La complejidad espacial del algoritmo de inserción directa es O(1), es decir, constante. Esto significa que el algoritmo no requiere memoria adicional proporcional al tamaño de la lista que se está ordenando.

El algoritmo de inserción directa opera directamente en la lista original, realizando comparaciones y movimientos de elementos entre los elementos existentes. No se requiere crear estructuras de datos adicionales ni asignar memoria adicional para almacenar el resultado del proceso de ordenamiento.

El espacio adicional utilizado por el algoritmo de inserción directa se limita a un número constante de variables auxiliares utilizadas para realizar las comparaciones y los movimientos de elementos. Estas variables ocupan una cantidad fija de memoria independientemente del tamaño de la lista.

Por lo tanto, la complejidad espacial del algoritmo de inserción directa es O(1), lo que implica que el consumo de memoria no aumenta a medida que aumenta el tamaño de la lista. Esto hace que el algoritmo de inserción directa sea eficiente en términos de espacio.

Binaria

Análisis de Eficiencia

El análisis de eficiencia del algoritmo de búsqueda binaria implica evaluar el número de operaciones requeridas en función del tamaño del conjunto de datos. La búsqueda binaria es un algoritmo eficiente para buscar un elemento en una lista ordenada.

Análisis de los Casos: mejor de los casos, caso medio y peor de los casos

Mejor caso:

El elemento buscado se encuentra exactamente en el centro de la lista. En el mejor caso, el algoritmo de búsqueda binaria encuentra el elemento deseado en el primer intento. Realiza una sola comparación para determinar si el elemento central es

igual al buscado y finaliza. La complejidad de tiempo en el mejor caso es O(1), ya que se encuentra rápidamente el elemento buscado.

Caso promedio:

El elemento buscado está ubicado en una posición aleatoria dentro de la lista. En promedio, el algoritmo de búsqueda binaria realiza aproximadamente log2(n) comparaciones para encontrar el elemento. En cada iteración, divide a la mitad el tamaño del conjunto de datos restante, descartando una porción significativa de los datos en cada paso. La complejidad de tiempo en el caso promedio es O(log n), donde n es el tamaño del conjunto de datos.

Peor caso:

El elemento buscado no está presente en la lista. En el peor caso, el algoritmo de búsqueda binaria recorre todo el conjunto de datos y no encuentra el elemento. Realiza log2(n) comparaciones antes de determinar que el elemento no está presente. La complejidad de tiempo en el peor caso es O(log n), ya que el número de comparaciones sigue siendo logarítmico.

Complejidad en el Tiempo

La búsqueda binaria tiene una complejidad de tiempo de O(log n), donde n es el tamaño del conjunto de datos. Esto implica que el tiempo de ejecución del algoritmo crece de manera logarítmica a medida que aumenta el tamaño de la lista. En cada iteración, el algoritmo divide a la mitad el tamaño del conjunto de datos restante. Esto se debe a que compara el elemento buscado con el elemento en la posición media y, en función del resultado, descarta la mitad de los datos restantes.

Como el tamaño del conjunto de datos se reduce a la mitad en cada iteración, el número de operaciones necesarias para encontrar el elemento buscado crece de manera logarítmica. La búsqueda binaria es especialmente útil cuando el tamaño del conjunto de datos es grande, ya que puede reducir significativamente el número de comparaciones en comparación con otros algoritmos de búsqueda lineal.

Complejidad Espacial

La complejidad espacial de la búsqueda binaria es O(1), es decir, constante. Esto se debe a que el algoritmo no requiere memoria adicional que dependa del tamaño del conjunto de datos. Solo se necesitan un número constante de variables para realizar las operaciones de búsqueda. El algoritmo de búsqueda binaria se realiza en el lugar, es decir, no se requiere una estructura de datos adicional para almacenar los datos. La búsqueda se realiza mediante la modificación de los índices y realizando comparaciones en el conjunto de datos original. Por lo tanto, la complejidad espacial es constante y no aumenta con el tamaño del conjunto de datos.

HeapSort

El algoritmo de ordenamiento HeapSort es un algoritmo de ordenamiento basado en estructuras de datos llamadas montículos o heaps. Un montículo es una estructura de datos de árbol binario completo en la que cada nodo padre es mayor o menor que sus nodos hijos, dependiendo de si es un montículo máximo o mínimo, respectivamente.

El algoritmo de HeapSort utiliza la propiedad de los montículos para construir un montículo máximo a partir de los elementos del arreglo y luego extrae el máximo repetidamente para obtener los elementos en orden ascendente. A grandes rasgos, el algoritmo se puede describir de la siguiente manera:

1. Construcción del montículo: Se construye un montículo máximo a partir del arreglo de elementos desordenados. Esto implica organizar los elementos en el arreglo de manera que se cumpla la propiedad del montículo.
2. Extracción del máximo: Se extrae el elemento máximo del montículo y se coloca al final del arreglo. Luego, se ajusta el montículo para mantener la propiedad del montículo.

3. Repetir: Se repite el paso 2 hasta que todos los elementos hayan sido extraídos del montículo. Al finalizar, el arreglo contendrá los elementos ordenados en forma ascendente.

El algoritmo de HeapSort tiene una complejidad temporal de O(n log n), lo que lo hace eficiente para grandes conjuntos de datos. Sin embargo, debido a las operaciones de construcción y ajuste del montículo, requiere memoria adicional y no es un algoritmo estable.

A continuación, te muestro una implementación del algoritmo HeapSort en Java:

```java
public class HeapSort {
    public static void sort(int[] arr) {
        int n = arr.length;

        // Construcción del montículo
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Extracción del máximo repetidamente
        for (int i = n - 1; i >= 0; i--) {
            // Mueve el máximo actual al final del arreglo
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Ajusta el montículo en el rango reducido
            heapify(arr, i, 0);
        }
    }
```

```java
private static void heapify(int[] arr, int n, int i) {
    int largest = i; // Inicializa el nodo raíz como el más grande
    int left = 2 * i + 1; // Índice del hijo izquierdo
    int right = 2 * i + 2; // Índice del hijo derecho

    // Si el hijo izquierdo es más grande que el nodo raíz
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // Si el hijo derecho es más grande que el nodo raíz o el hijo izquierdo
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // Si el mayor no es el nodo raíz
    if (largest != i) {
        // Intercambia el nodo raíz con el mayor
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Ajusta recursivamente el subárbol afectado
        heapify(arr, n, largest);
    }
}
```

```java
public static void main(String[] args) {
    int[] arr = { 12, 11, 13, 5, 6, 7 };
    sort(arr);
    System.out.println("Arreglo ordenado:");
    for (int num : arr) {
        System.out.print(num + " ");
    }
}
```

Analisis de eficiencia

El algoritmo HeapSort tiene una eficiencia temporal de O(n log n) en el peor y mejor de los casos, donde "n" es el número de elementos en el arreglo a ordenar.

El proceso de construcción del montículo tiene una complejidad de O(n), ya que se requiere recorrer todos los elementos del arreglo y realizar operaciones de ajuste para mantener la propiedad del montículo. Este paso se realiza una sola vez al inicio del algoritmo.

La etapa de extracción del máximo se repite "n" veces, ya que se extrae el máximo y se ajusta el montículo en cada iteración. El ajuste del montículo (función **heapify**) tiene una complejidad de O(log n), ya que implica recorrer el árbol binario hasta la hoja más profunda, lo cual se hace a través de comparaciones y movimientos de elementos.

En general, el rendimiento de HeapSort es eficiente debido a su complejidad O(n log n), lo que lo hace adecuado para ordenar grandes conjuntos de datos. Sin embargo, en comparación con otros algoritmos de ordenamiento como QuickSort y MergeSort, HeapSort puede ser un poco más lento debido a las operaciones adicionales requeridas para construir y ajustar el montículo en cada iteración. Además, HeapSort no es un algoritmo estable, lo que significa que puede cambiar el orden relativo de elementos con claves iguales.

Analisis de casos

En el mejor de los casos: En el mejor de los casos, el algoritmo HeapSort tiene una eficiencia temporal de O(n log n). Esto ocurre cuando el arreglo ya está previamente ordenado en orden ascendente o descendente. Durante la construcción del montículo, no se realizarán ajustes ya que cada elemento cumple con la propiedad del montículo. Luego, en la etapa de extracción del máximo, se realizarán las comparaciones y ajustes necesarios, pero como el montículo ya está en su forma final, el número de operaciones requeridas se reduce significativamente. Por lo tanto, en el mejor de los casos, HeapSort muestra un rendimiento óptimo de O(n log n).

En el peor de los casos: En el peor de los casos, el algoritmo HeapSort también tiene una eficiencia temporal de O(n log n). Esto ocurre cuando el arreglo está completamente desordenado y los elementos deben ser reorganizados en el montículo en cada iteración. Durante la construcción del montículo, se requieren O(n) operaciones para ajustar los elementos y lograr la propiedad del montículo. Luego, en la etapa de extracción del máximo, se realizan O(log n) comparaciones y ajustes en cada iteración, y esta etapa se repite "n" veces. En total, el número de operaciones requeridas es proporcional a n log n, lo que resulta en una complejidad temporal de O(n log n) en el peor de los casos.

En un caso medio: En un caso medio, HeapSort sigue teniendo una eficiencia temporal de O(n log n). En general, el rendimiento del algoritmo se mantiene constante y no depende de la distribución particular de los elementos en el arreglo. La construcción del montículo requiere O(n) operaciones y la etapa de extracción del máximo requiere O(log n) operaciones en cada iteración. Aunque puede haber algunas variaciones en la cantidad de operaciones realizadas debido a la distribución de los datos, en promedio, la complejidad sigue siendo O(n log n). Por lo tanto, HeapSort mantiene su rendimiento eficiente en un caso medio.

Complejidad en el tiempo y complejidad espacial

La complejidad en el tiempo del algoritmo HeapSort es de O(n log n) en todos los casos, incluyendo el mejor de los casos, el peor de los casos y los casos promedio. Esto significa que el tiempo de ejecución del algoritmo crece proporcionalmente al tamaño del arreglo (n) multiplicado por el logaritmo del tamaño del arreglo.

En cuanto a la complejidad espacial, HeapSort tiene una complejidad de espacio de O(1) en el peor de los casos, el mejor de los casos y los casos promedio. Esto se debe a que HeapSort ordena los elementos en el mismo arreglo que se le proporciona, sin requerir estructuras de datos auxiliares adicionales que crezcan proporcionalmente al tamaño del arreglo. Por lo tanto, el uso de memoria adicional es constante y no depende del tamaño del arreglo.

Quicksort recursivo

El algoritmo de ordenamiento Quicksort es un algoritmo de ordenamiento recursivo basado en la técnica de "dividir y conquistar". Fue desarrollado por Tony Hoare en 1959 y es ampliamente utilizado debido a su eficiencia en la práctica.

La idea principal detrás del algoritmo Quicksort es seleccionar un elemento del arreglo, denominado "pivote", y particionar el arreglo en dos subarreglos: uno que contiene elementos menores que el pivote y otro que contiene elementos mayores que el pivote. Luego, se aplica el mismo proceso de manera recursiva a cada subarreglo hasta que los subarreglos sean lo suficientemente pequeños y estén ordenados. Finalmente, se combinan los subarreglos ordenados para obtener el arreglo ordenado completo.

Aquí tienes una implementación básica del algoritmo Quicksort en Java:

```java
public class QuickSort {
    public static void sort(int[] arr) {
        quicksort(arr, 0, arr.length - 1);
    }

    private static void quicksort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high); // Índice de partición
            quicksort(arr, low, pi - 1); // Ordenar subarreglo inferior
            quicksort(arr, pi + 1, high); // Ordenar subarreglo superior
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // Seleccionar el pivote (último elemento)
        int i = low - 1; // Índice del elemento más pequeño

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j); // Intercambiar elementos
            }
        }
```

```
        swap(arr, i + 1, high); // Colocar el pivote en la posición correcta
        return i + 1; // Retornar el índice de partición
    }

    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void main(String[] args) {
        int[] arr = { 6, 2, 8, 1, 4, 9 };
        sort(arr);
        System.out.println("Arreglo ordenado:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Analisis de casos

En el mejor de los casos: En el mejor de los casos, el algoritmo Quicksort tiene una eficiencia temporal de O(n log n). Esto ocurre cuando el pivote seleccionado divide el arreglo en dos subarreglos de tamaño aproximadamente igual en cada recursión. En este escenario ideal, el algoritmo logra realizar particiones balanceadas y se logra el mejor rendimiento posible. Cada nivel de recursión tiene un costo de O(n) debido a las comparaciones y los intercambios, y dado que el árbol de recursión tiene una altura de log n, el tiempo total de ejecución es O(n log n).

En el peor de los casos: En el peor de los casos, el algoritmo Quicksort tiene una eficiencia temporal de O(n^2). Esto ocurre cuando el pivote seleccionado siempre es el elemento más pequeño o el más grande en cada partición. Como resultado, el algoritmo no logra dividir el arreglo de manera equilibrada y se crea una partición muy desigual en cada paso. En este caso, la recursión tiene que procesar n

elementos en cada nivel, y como el árbol de recursión tiene una altura de n, el tiempo total de ejecución es O(n^2).

En un caso medio: En un caso medio, el algoritmo Quicksort tiene una eficiencia temporal esperada de O(n log n). En promedio, el pivote seleccionado divide el arreglo en subarreglos de tamaño aproximadamente igual en cada recursión. Aunque puede haber variaciones en la distribución de los elementos y en el rendimiento de las particiones en cada ejecución, se ha demostrado que, en promedio, el número de comparaciones y los intercambios necesarios para ordenar el arreglo siguen siendo proporcionales a n log n. Por lo tanto, en un caso medio, Quicksort muestra un rendimiento eficiente.

Complejidad en el tiempo y complejidad espacial

La complejidad en el tiempo del algoritmo Quicksort es de O(n log n) en promedio y en el mejor de los casos. Esto significa que el tiempo de ejecución del algoritmo crece proporcionalmente al tamaño del arreglo (n) multiplicado por el logaritmo del tamaño del arreglo.

Sin embargo, en el peor de los casos, la complejidad en el tiempo puede ser de O(n^2). Esto ocurre cuando el pivote seleccionado siempre es el elemento más pequeño o el más grande en cada partición, lo que resulta en particiones muy desequilibradas. En este caso, el tiempo de ejecución del algoritmo puede aumentar significativamente.

La complejidad espacial del algoritmo Quicksort es de O(log n) en la pila de llamadas debido a la recursión. En cada nivel de recursión, se necesita un marco de pila adicional para realizar las llamadas recursivas y almacenar los parámetros y variables locales. El número de niveles de recursión en el árbol de llamadas es proporcional al logaritmo del tamaño del arreglo.

Además del uso de la pila de llamadas, el algoritmo Quicksort generalmente no requiere memoria adicional proporcional al tamaño del arreglo. En lugar de eso, el algoritmo opera directamente sobre el arreglo existente y realiza intercambios en su lugar, lo que lo convierte en un algoritmo de ordenamiento in situ (en el lugar).

Algortimos de ordenamiento externos

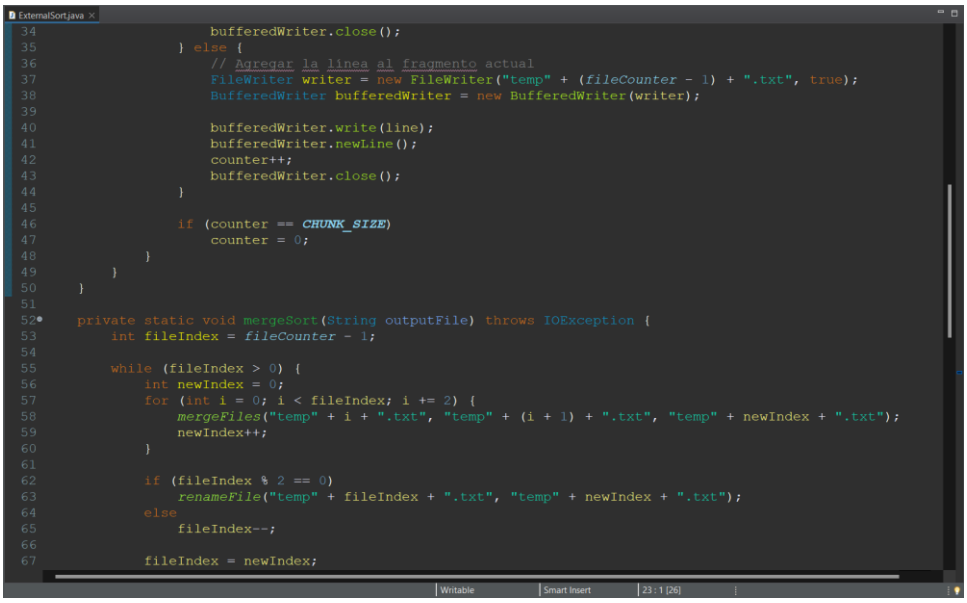
Introducción:

Los algoritmos de ordenamiento externo son utilizados cuando los datos a ordenar no caben completamente en la memoria principal (RAM) y deben ser almacenados en dispositivos de almacenamiento secundario, como discos duros o cintas magnéticas. Estos algoritmos manejan la limitación de acceso a los datos en estos dispositivos, optimizando las operaciones de lectura y escritura.

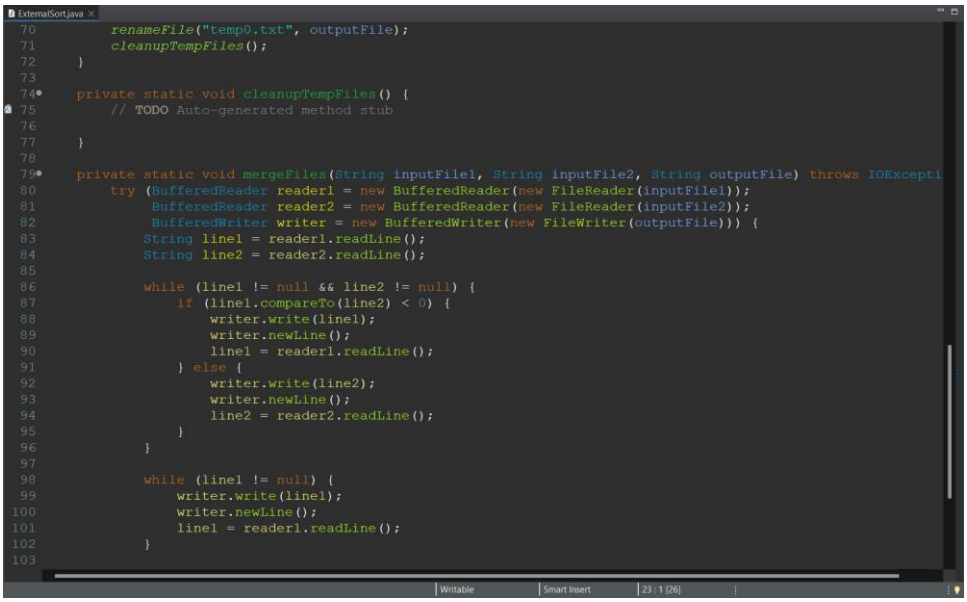
Ejemplo (Código):

ExternalSort.java

```java
package Tarea;
import java.io.*;

public class ExternalSort {


    private static final int CHUNK_SIZE = 1000; // Tamaño de los fragmentos en memoria
    private static int fileCounter = 0;

    public static void externalSort(String inputFile, String outputFile) {
        try {
            splitFile(inputFile); // Dividir el archivo en fragmentos más pequeños
            mergeSort(outputFile); // Ordenar y fusionar los fragmentos
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void splitFile(String inputFile) throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {
            String line;
            int counter = 0;

            while ((line = reader.readLine()) != null) {
                if (counter == 0) {
                    // Crear un nuevo archivo para el fragmento
                    FileWriter writer = new FileWriter("temp" + fileCounter + ".txt");
                    BufferedWriter bufferedWriter = new BufferedWriter(writer);
                    fileCounter++;

                    bufferedWriter.write(line);
                    bufferedWriter.newLine();
                    counter++;
                    bufferedWriter.close();
```



```
ExternalSort.java
34                      bufferedWriter.close();
35                  } else {
36                      // Agregar la línea al fragmento actual
37                      FileWriter writer = new FileWriter("temp" + (fileCounter - 1) + ".txt", true);
38                      BufferedWriter bufferedWriter = new BufferedWriter(writer);
39
40                      bufferedWriter.write(line);
41                      bufferedWriter.newLine();
42                      counter++;
43                      bufferedWriter.close();
44                  }
45
46                  if (counter == CHUNK_SIZE)
47                      counter = 0;
48              }
49          }
50      }
51
52      private static void mergeSort(String outputFile) throws IOException {
53          int fileIndex = fileCounter - 1;
54
55          while (fileIndex > 0) {
56              int newIndex = 0;
57              for (int i = 0; i < fileIndex; i += 2) {
58                  mergeFiles("temp" + i + ".txt", "temp" + (i + 1) + ".txt", "temp" + newIndex + ".txt");
59                  newIndex++;
60              }
61
62              if (fileIndex % 2 == 0)
63                  renameFile("temp" + fileIndex + ".txt", "temp" + newIndex + ".txt");
64              else
65                  fileIndex--;
66
67              fileIndex = newIndex;
                Writable | Smart Insert | 23 : 1 [26]
```



```
ExternalSort.java
70          renameFile("temp0.txt", outputFile);
71          cleanupTempFiles();
72      }
73
74      private static void cleanupTempFiles() {
75          // TODO Auto-generated method stub
76
77      }
78
79      private static void mergeFiles(String inputFile1, String inputFile2, String outputFile) throws IOExcepti
80          try (BufferedReader reader1 = new BufferedReader(new FileReader(inputFile1));
81               BufferedReader reader2 = new BufferedReader(new FileReader(inputFile2));
82               BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
83              String line1 = reader1.readLine();
84              String line2 = reader2.readLine();
85
86              while (line1 != null && line2 != null) {
87                  if (line1.compareTo(line2) < 0) {
88                      writer.write(line1);
89                      writer.newLine();
90                      line1 = reader1.readLine();
91                  } else {
92                      writer.write(line2);
93                      writer.newLine();
94                      line2 = reader2.readLine();
95                  }
96              }
97
98              while (line1 != null) {
99                  writer.write(line1);
100                 writer.newLine();
101                 line1 = reader1.readLine();
102             }
103
                Writable | Smart Insert | 23 : 1 [26]
```

```java
            String line1 = reader1.readLine();
            String line2 = reader2.readLine();

            while (line1 != null && line2 != null) {
                if (line1.compareTo(line2) < 0) {
                    writer.write(line1);
                    writer.newLine();
                    line1 = reader1.readLine();
                } else {
                    writer.write(line2);
                    writer.newLine();
                    line2 = reader2.readLine();
                }
            }

            while (line1 != null) {
                writer.write(line1);
                writer.newLine();
                line1 = reader1.readLine();
            }

            while (line2 != null) {
                writer.write(line2);
                writer.newLine();
                line2 = reader2.readLine();
            }
        }
    }

    private static void renameFile(String oldName, String newName) {
        File file = new File(oldName);
        file.renameTo(new File(newName));
    }
}
```

Análisis de Eficiencia:

En resumen, el análisis de eficiencia del algoritmo de ordenamiento externo involucra considerar la complejidad temporal, el consumo de memoria y el acceso a disco. La eficiencia del algoritmo dependerá de la implementación específica, las características de los datos y los recursos disponibles en el sistema. Es importante realizar pruebas y evaluaciones empíricas para determinar el rendimiento real del algoritmo en un entorno específico.

Análisis de los Casos:

- ◊ **Mejor caso:**

El mejor caso ocurre cuando los datos ya están ordenados o tienen un patrón que facilita la fase de fusión del algoritmo de ordenamiento externo. En este caso, el algoritmo puede realizar menos operaciones de lectura y escritura en disco, lo que reduce el tiempo de ejecución. La fase de división de los datos puede requerir la misma cantidad de operaciones que en el peor caso, pero la fase de fusión será más eficiente. En general, el mejor caso tendría una complejidad temporal cercana

a O(N log(fileCounter)), donde N es el número de elementos a ordenar y fileCounter es el número de fragmentos generados.

◊ **Caso medio:**

El caso medio considera una distribución aleatoria de los datos. No hay un patrón definido ni una secuencia específica. En este caso, el algoritmo de ordenamiento externo puede realizar una cantidad moderada de operaciones de lectura y escritura en disco. El tiempo de ejecución dependerá de la distribución y cantidad de datos, así como de la cantidad de memoria disponible para el proceso. La complejidad temporal promedio sería O(N log(fileCounter)).

◊ **Peor caso:**

El peor caso ocurre cuando los datos están inversamente ordenados o tienen un patrón que dificulta la fase de fusión del algoritmo de ordenamiento externo. En este caso, el algoritmo puede requerir una mayor cantidad de operaciones de lectura y escritura en disco para reorganizar los datos durante las fases de división y fusión. La fase de división de los datos puede requerir la misma cantidad de operaciones que en el mejor caso, pero la fase de fusión será menos eficiente. En general, el peor caso tendría una complejidad temporal cercana a O(N log(fileCounter)), donde N es el número de elementos a ordenar y fileCounter es el número de fragmentos generados.

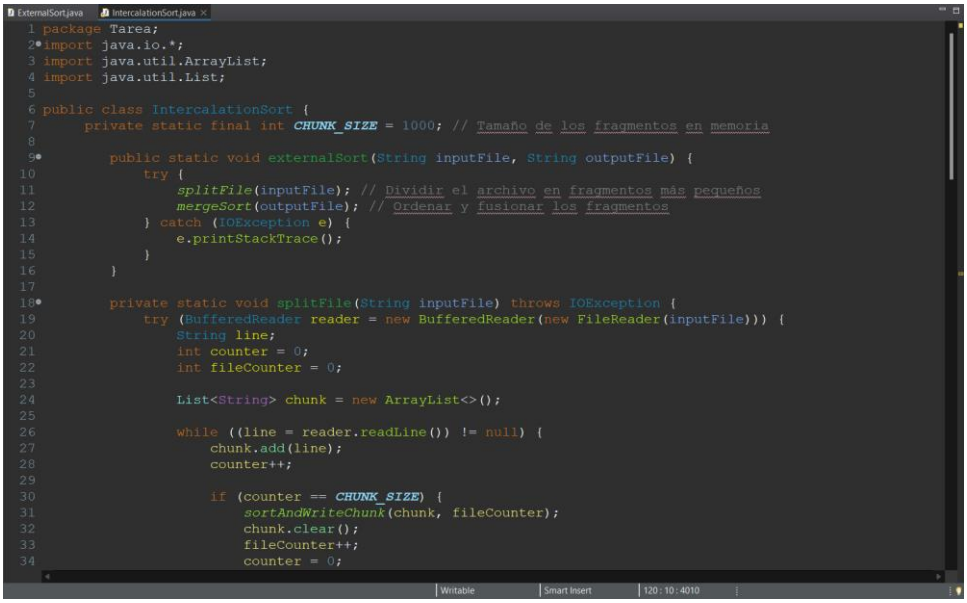
intercalacion

Introducción:

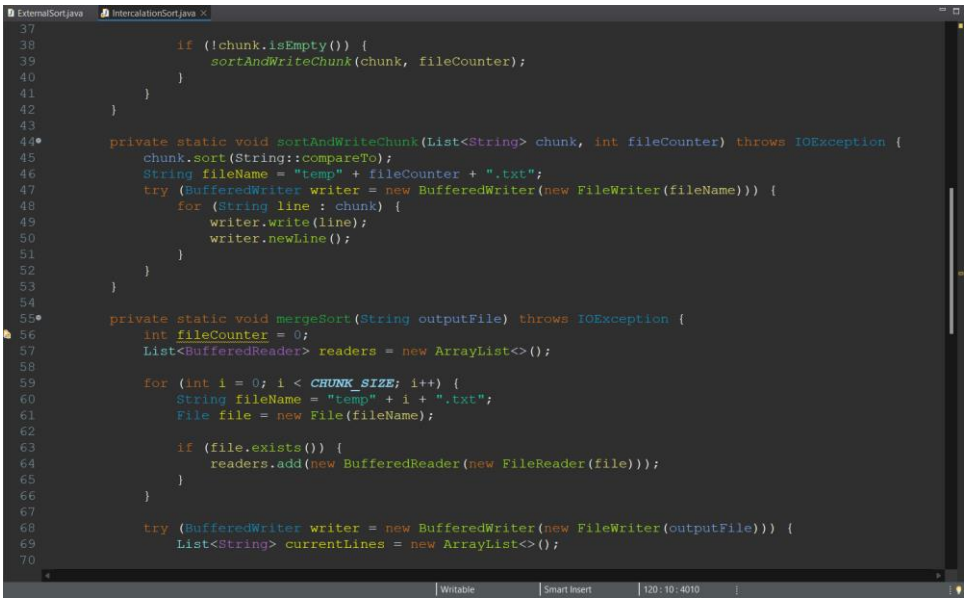
El algoritmo de ordenamiento por intercalación es utilizado para ordenar grandes conjuntos de datos que no caben en memoria principal. Se divide el conjunto en subconjuntos más pequeños que se ordenan individualmente y se guardan en archivos temporales. Luego, se realiza la intercalación, que consiste en combinar y ordenar los subconjuntos en uno solo. Este proceso se repite hasta obtener un conjunto final ordenado. El algoritmo minimiza el acceso a disco y utiliza la memoria

eficientemente, pero su rendimiento puede verse afectado por la velocidad de acceso a disco y el tamaño de los subconjuntos en memoria. En resumen, el algoritmo de intercalación es eficiente para ordenar grandes conjuntos de datos, garantizando un resultado final ordenado.

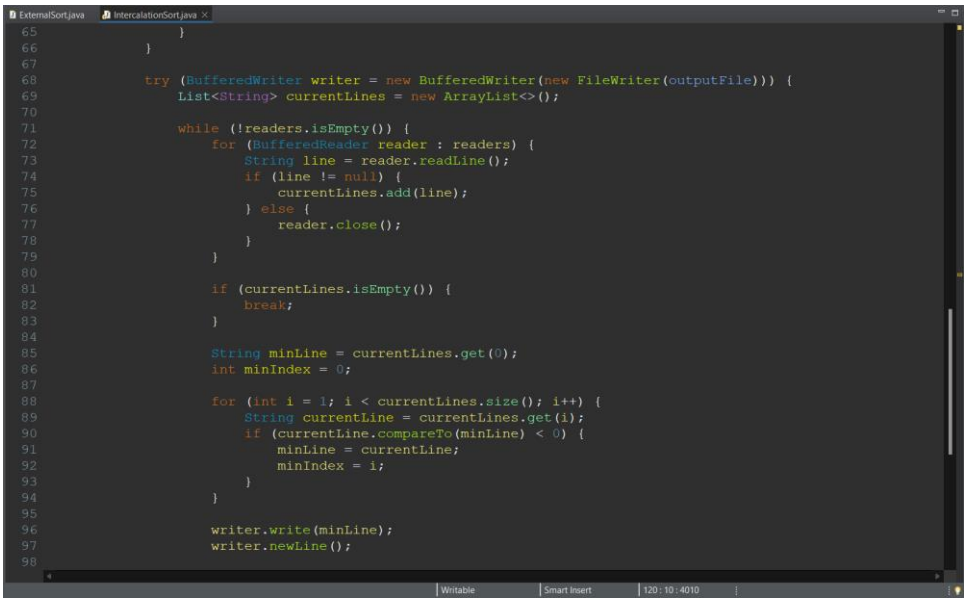
Ejemplo (Código):



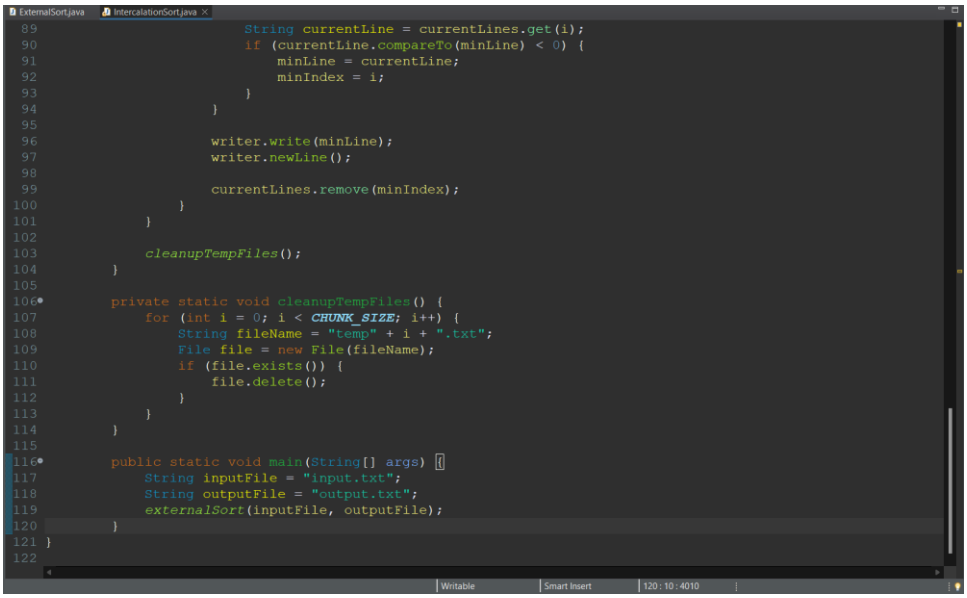
```java
package Tarea;
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class IntercalationSort {
    private static final int CHUNK_SIZE = 1000; // Tamaño de los fragmentos en memoria

    public static void externalSort(String inputFile, String outputFile) {
        try {
            splitFile(inputFile); // Dividir el archivo en fragmentos más pequeños
            mergeSort(outputFile); // Ordenar y fusionar los fragmentos
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void splitFile(String inputFile) throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {
            String line;
            int counter = 0;
            int fileCounter = 0;

            List<String> chunk = new ArrayList<>();

            while ((line = reader.readLine()) != null) {
                chunk.add(line);
                counter++;

                if (counter == CHUNK_SIZE) {
                    sortAndWriteChunk(chunk, fileCounter);
                    chunk.clear();
                    fileCounter++;
                    counter = 0;
```



```java
                if (!chunk.isEmpty()) {
                    sortAndWriteChunk(chunk, fileCounter);
                }
            }
        }

    private static void sortAndWriteChunk(List<String> chunk, int fileCounter) throws IOException {
        chunk.sort(String::compareTo);
        String fileName = "temp" + fileCounter + ".txt";
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {
            for (String line : chunk) {
                writer.write(line);
                writer.newLine();
            }
        }
    }

    private static void mergeSort(String outputFile) throws IOException {
        int fileCounter = 0;
        List<BufferedReader> readers = new ArrayList<>();

        for (int i = 0; i < CHUNK_SIZE; i++) {
            String fileName = "temp" + i + ".txt";
            File file = new File(fileName);

            if (file.exists()) {
                readers.add(new BufferedReader(new FileReader(file)));
            }
        }

        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
            List<String> currentLines = new ArrayList<>();
```



```
65                  }
66              }
67
68              try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
69                  List<String> currentLines = new ArrayList<>();
70
71                  while (!readers.isEmpty()) {
72                      for (BufferedReader reader : readers) {
73                          String line = reader.readLine();
74                          if (line != null) {
75                              currentLines.add(line);
76                          } else {
77                              reader.close();
78                          }
79                      }
80
81                      if (currentLines.isEmpty()) {
82                          break;
83                      }
84
85                      String minLine = currentLines.get(0);
86                      int minIndex = 0;
87
88                      for (int i = 1; i < currentLines.size(); i++) {
89                          String currentLine = currentLines.get(i);
90                          if (currentLine.compareTo(minLine) < 0) {
91                              minLine = currentLine;
92                              minIndex = i;
93                          }
94                      }
95
96                      writer.write(minLine);
97                      writer.newLine();
98
```



```
89                          String currentLine = currentLines.get(i);
90                          if (currentLine.compareTo(minLine) < 0) {
91                              minLine = currentLine;
92                              minIndex = i;
93                          }
94                      }
95
96                      writer.write(minLine);
97                      writer.newLine();
98
99                      currentLines.remove(minIndex);
100                 }
101             }
102
103             cleanupTempFiles();
104         }
105
106         private static void cleanupTempFiles() {
107             for (int i = 0; i < CHUNK_SIZE; i++) {
108                 String fileName = "temp" + i + ".txt";
109                 File file = new File(fileName);
110                 if (file.exists()) {
111                     file.delete();
112                 }
113             }
114         }
115
116         public static void main(String[] args) {
117             String inputFile = "input.txt";
118             String outputFile = "output.txt";
119             externalSort(inputFile, outputFile);
120         }
121 }
122
```

Análisis de Eficiencia:

El algoritmo de ordenamiento por intercalación es eficiente para manejar grandes conjuntos de datos que no caben en memoria principal. Su rendimiento depende del tamaño del conjunto de datos, el número de operaciones de intercalación y la eficiencia del algoritmo de ordenamiento interno utilizado. En general, es una opción efectiva para el ordenamiento externo, minimizando el acceso al almacenamiento secundario y utilizando eficientemente la memoria disponible.

Análisis de los Casos:

Mejor caso:

- El mejor caso ocurre cuando el conjunto de datos ya está parcialmente ordenado.
- En este caso, durante la división inicial, los fragmentos generados pueden estar más cerca de estar ordenados, lo que reduce la cantidad de operaciones de intercalación necesarias.
- La etapa de intercalación se vuelve más eficiente, ya que los subconjuntos ordenados se combinan de manera más directa y requieren menos comparaciones y movimientos de datos.
- Como resultado, el tiempo de ejecución en el mejor caso puede ser menor en comparación con los casos promedio o peor.

Caso medio:

- El caso medio se refiere a una distribución típica de los datos en el conjunto a ordenar.
- En este caso, la eficiencia del algoritmo de ordenamiento por intercalación dependerá de factores como el tamaño del conjunto de datos, la distribución de los elementos y la eficiencia del algoritmo de ordenamiento interno utilizado.
- Durante la división inicial, los fragmentos generados pueden variar en tamaño y orden, lo que puede requerir una cantidad moderada de operaciones de intercalación en la etapa de fusión.
- El tiempo de ejecución en el caso medio será proporcional al tamaño del conjunto de datos y a la cantidad de operaciones de intercalación realizadas.

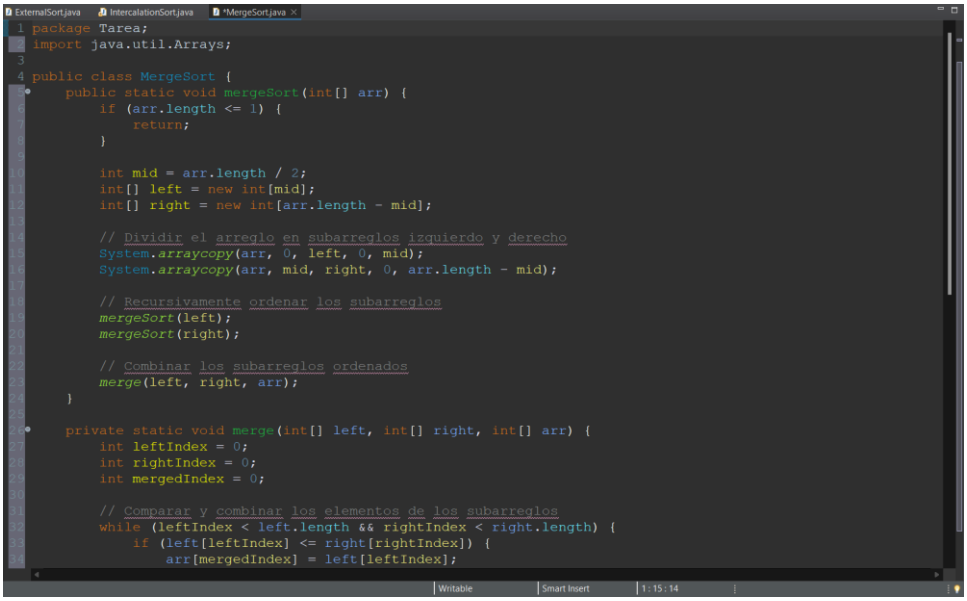
Peor caso:

- El peor caso ocurre cuando el conjunto de datos está desordenado de manera que requiere un alto número de operaciones de intercalación.
- Durante la división inicial, los fragmentos generados pueden ser de tamaño similar pero desordenados, lo que requerirá una cantidad máxima de operaciones de intercalación en la etapa de fusión.
- La cantidad de operaciones de intercalación en el peor caso está relacionada con el tamaño del conjunto de datos y puede aumentar significativamente a medida que el tamaño del conjunto de datos crece.
- Como resultado, el tiempo de ejecución en el peor caso será mayor en comparación con los casos medio y mejor.

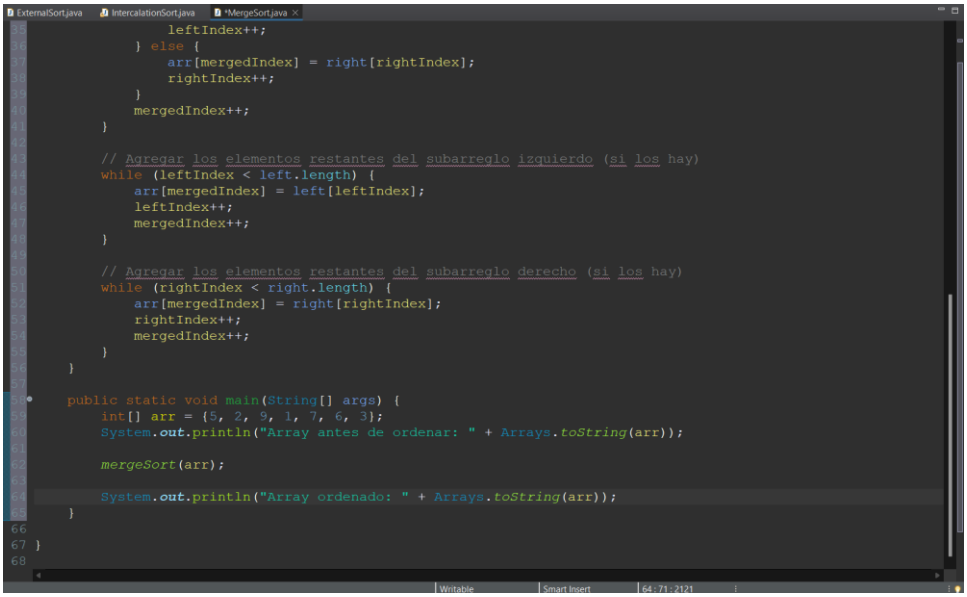
Mezcla directa

El algoritmo de ordenamiento por mezcla directa, conocido como merge sort, es un método eficiente para ordenar conjuntos de datos. Se divide el conjunto en subconjuntos más pequeños, se ordenan individualmente y luego se fusionan para obtener un conjunto final ordenado. Es útil cuando se trabaja con conjuntos grandes que no caben en memoria, ya que minimiza el acceso a disco y utiliza eficientemente la memoria disponible. El proceso consta de dividir los datos en subconjuntos, fusionarlos recursivamente y comparar y ordenar los elementos durante la fusión. La eficiencia se basa en la capacidad de dividir y combinar eficientemente los subconjuntos ordenados. Lo que lo hace adecuado para conjuntos de datos grandes. En resumen, el merge sort es un método eficiente basado en dividir y conquistar, adecuado para ordenar conjuntos grandes y aprovechar eficientemente los recursos disponibles.

Ejemplo (Código):



```java
package Tarea;
import java.util.Arrays;

public class MergeSort {
    public static void mergeSort(int[] arr) {
        if (arr.length <= 1) {
            return;
        }

        int mid = arr.length / 2;
        int[] left = new int[mid];
        int[] right = new int[arr.length - mid];

        // Dividir el arreglo en subarreglos izquierdo y derecho
        System.arraycopy(arr, 0, left, 0, mid);
        System.arraycopy(arr, mid, right, 0, arr.length - mid);

        // Recursivamente ordenar los subarreglos
        mergeSort(left);
        mergeSort(right);

        // Combinar los subarreglos ordenados
        merge(left, right, arr);
    }

    private static void merge(int[] left, int[] right, int[] arr) {
        int leftIndex = 0;
        int rightIndex = 0;
        int mergedIndex = 0;

        // Comparar y combinar los elementos de los subarreglos
        while (leftIndex < left.length && rightIndex < right.length) {
            if (left[leftIndex] <= right[rightIndex]) {
                arr[mergedIndex] = left[leftIndex];
```



```java
                leftIndex++;
            } else {
                arr[mergedIndex] = right[rightIndex];
                rightIndex++;
            }
            mergedIndex++;
        }

        // Agregar los elementos restantes del subarreglo izquierdo (si los hay)
        while (leftIndex < left.length) {
            arr[mergedIndex] = left[leftIndex];
            leftIndex++;
            mergedIndex++;
        }

        // Agregar los elementos restantes del subarreglo derecho (si los hay)
        while (rightIndex < right.length) {
            arr[mergedIndex] = right[rightIndex];
            rightIndex++;
            mergedIndex++;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 7, 6, 3};
        System.out.println("Array antes de ordenar: " + Arrays.toString(arr));

        mergeSort(arr);

        System.out.println("Array ordenado: " + Arrays.toString(arr));
    }

}
```

Análisis de Eficiencia:

El algoritmo de ordenamiento por intercalación es eficiente para manejar grandes conjuntos de datos que no caben en memoria principal. Su rendimiento depende del tamaño del conjunto de datos, el número de operaciones de intercalación y la eficiencia del algoritmo de ordenamiento interno utilizado. En general, es una opción

efectiva para el ordenamiento externo, minimizando el acceso al almacenamiento secundario y utilizando eficientemente la memoria disponible.

Análisis de los Casos:

- **Mejor caso:**

El mejor caso ocurre cuando el conjunto de datos ya está ordenado. En este caso, el algoritmo de mezcla directa aún realiza las divisiones recursivas y las fusiones, pero no necesita realizar las comparaciones y fusiones en el paso de fusión. La eficiencia en el mejor caso sigue siendo O(n log n) debido a las divisiones recursivas, pero el factor constante es menor ya que no se realizan muchas comparaciones y fusiones.

- **Caso promedio:**

El caso promedio del algoritmo de mezcla directa también tiene una eficiencia de O(n log n). A medida que se realiza la división recursiva, el conjunto de datos se divide en subconjuntos más pequeños hasta que cada subconjunto contenga solo un elemento. Luego, en el paso de fusión, los subconjuntos se combinan en pares y se comparan y fusionan en un conjunto final ordenado. Este proceso se repite hasta que se haya obtenido un único conjunto ordenado. La eficiencia se mantiene constante debido a la naturaleza recursiva del algoritmo y la división equitativa de los subconjuntos.

- **Peor caso:**

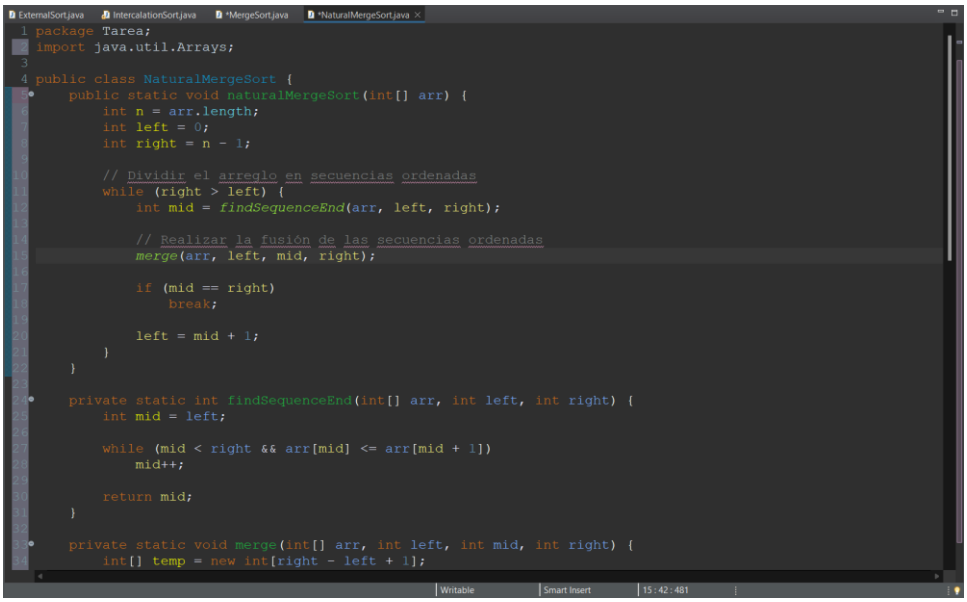
El peor caso ocurre cuando el conjunto de datos está completamente desordenado. En este caso, el algoritmo de mezcla directa realiza todas las divisiones recursivas y las fusiones correspondientes. La eficiencia sigue siendo O(n log n), ya que, a pesar de las muchas comparaciones y fusiones requeridas, la división recursiva sigue dividiendo el conjunto de datos en subconjuntos más pequeños hasta que contengan solo un elemento.

Mezcla natural

El algoritmo de ordenamiento por mezcla natural es un método eficiente que utiliza la técnica de mezcla para ordenar conjuntos de datos. Aprovecha las secuencias ordenadas presentes en el conjunto de datos para reducir la cantidad de comparaciones y fusiones necesarias.

El proceso de ordenamiento comienza dividiendo el conjunto de datos en subsecuencias ordenadas. Luego, estas subsecuencias se fusionan en secuencias más grandes hasta obtener un conjunto completamente ordenado. Durante la fusión, se comparan los elementos y se colocan en orden en la secuencia resultante.

Ejemplo (Código):



```java
        int[] temp = new int[right - left + 1];
        int i = left;
        int j = mid + 1;
        int k = 0;

        // Fusionar las dos secuencias ordenadas
        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k] = arr[i];
                i++;
            } else {
                temp[k] = arr[j];
                j++;
            }
            k++;
        }

        // Copiar los elementos restantes de la primera secuencia
        while (i <= mid) {
            temp[k] = arr[i];
            i++;
            k++;
        }

        // Copiar los elementos restantes de la segunda secuencia
        while (j <= right) {
            temp[k] = arr[j];
            j++;
            k++;
        }

        // Copiar los elementos ordenados al arreglo original
        for (int m = 0; m < temp.length; m++) {
            arr[left + m] = temp[m];
```

```java
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 6, 3};

        System.out.println("Arreglo original: " + Arrays.toString(arr));
        naturalMergeSort(arr);
        System.out.println("Arreglo ordenado: " + Arrays.toString(arr));
    }
}
```

Análisis de Eficiencia:

El análisis de eficiencia del algoritmo de ordenamiento mezcla natural (natural merge sort) se basa en la cantidad y longitud de las secuencias ordenadas presentes en el conjunto de datos.

En general, el algoritmo de mezcla natural es eficiente cuando se trabaja con conjuntos de datos parcialmente ordenados, ya que aprovecha las secuencias ordenadas existentes para reducir el número de comparaciones y operaciones de fusión. Sin embargo, en casos donde el conjunto de datos está completamente desordenado, el algoritmo puede tener un rendimiento similar al merge sort convencional.

Análisis de los Casos:

- **Mejor caso:**

El mejor caso ocurre cuando el conjunto de datos ya está completamente ordenado en secuencias. En este caso, no se requiere ninguna comparación adicional ni operaciones de fusión, ya que las secuencias ya están en orden y simplemente se concatenan. La eficiencia en el mejor caso es lineal, O(n), donde "n" es el tamaño del conjunto de datos.

- **Caso medio:**

El caso medio ocurre cuando las secuencias ordenadas están distribuidas de manera más uniforme en el conjunto de datos. Esto significa que hay algunas secuencias ordenadas dispersas y de longitudes similares. En el caso medio, el algoritmo de mezcla natural realiza comparaciones y operaciones de fusión para combinar las secuencias. La eficiencia en el caso medio es O(n log n), similar al merge sort convencional.

- **Peor caso:**

El peor caso ocurre cuando todas las secuencias ordenadas están juntas y tienen longitudes significativas. En este caso, el algoritmo de mezcla natural debe fusionar todas las secuencias en cada iteración, lo que implica un mayor número de comparaciones y operaciones de fusión. La eficiencia en el peor caso es O(n log n), similar al merge sort convencional.