Tecnológico Nacional de México
Campus Orizaba

Estructura de Datos

Ingeniería en Sistemas Computacionales

Tema 5:
Métodos de Ordenamiento

Integrantes:

Castillo Solís Luis Ángel – 21010932
Flores Domínguez Ángel Gabriel – 21010951
Muñoz Hernández Vania Lizeth – 21011009
Romero Ovando Karyme Michelle – 21011037

Grupo:
3g2B

Fecha de entrega: 30 / Mayo /2023

1. Introducción

Los algoritmos de ordenamiento son una parte fundamental de las estructuras de datos, utilizados para organizar conjuntos de datos de manera eficiente y facilitar su posterior búsqueda y manipulación. Existen dos tipos principales de algoritmos de ordenamiento: los algoritmos de ordenamiento internos y los algoritmos de ordenamiento externos. Cada tipo está diseñado para manejar diferentes tamaños de conjuntos de datos y utilizar diferentes estrategias de procesamiento.

Los algoritmos de ordenamiento internos son aquellos que se ejecutan en la memoria principal de una computadora, también conocida como memoria RAM. Son adecuados para conjuntos de datos que pueden caber completamente en la memoria principal y se caracterizan por ser rápidos y eficientes en términos de tiempo de ejecución.
Los algoritmos de ordenamiento externos son utilizados cuando la cantidad de datos a ordenar excede la capacidad de la memoria principal de una computadora. Están diseñados para minimizar los accesos al dispositivo de almacenamiento y optimizar la eficiencia en términos de tiempo de ejecución.

Ambos tipos de algoritmos desempeñan un papel crucial en la manipulación y procesamiento eficiente de datos en la programación.

2. Competencia específica

Conoce, comprende y aplica eficientemente estructuras de datos, métodos de ordenamiento y búsqueda para la optimización del rendimiento de soluciones a problemas del mundo real.

3. Marco Teórico

Temas:

5.1 Algoritmos de Ordenamiento Internos.

Los algoritmos de ordenamiento interno se refieren a aquellos métodos de ordenamiento que operan en la memoria primaria, es decir, la memoria de trabajo o RAM de un sistema. Estos algoritmos se utilizan cuando los datos a ordenar se encuentran en la memoria principal y se asume que el tiempo necesario para acceder a cualquier elemento de la lista es el mismo.

Estos algoritmos son comúnmente utilizados cuando se trabaja con listas simples, es decir, listas que contienen elementos de un solo tipo de datos. La principal característica de los algoritmos de ordenamiento interno es que realizan la ordenación de los elementos mientras se trabaja de forma preliminar con la lista, ya sea insertando nuevos datos o inicializando la lista.

La ordenación interna tiene la ventaja de ser más eficiente en tiempo de ejecución en comparación con los algoritmos de ordenamiento externo, que se utilizan cuando los datos a ordenar superan la capacidad de la memoria principal y deben almacenarse en dispositivos de almacenamiento secundarios, como discos duros. Al trabajar directamente en la memoria RAM, los algoritmos de ordenamiento interno pueden acceder rápidamente a los datos y realizar las operaciones de intercambio necesarias para lograr la ordenación deseada.

Es importante tener en cuenta que los algoritmos de ordenamiento interno son adecuados para conjuntos de datos de tamaño moderado o pequeño, ya que su rendimiento puede deteriorarse significativamente en conjuntos de datos muy grandes debido a la limitada capacidad de la memoria principal. En esos casos, los algoritmos de ordenamiento externo son más apropiados. Existen diversos algoritmos de ordenamiento interno ampliamente utilizados, por ejemplo:

- Burbuja con señal:

  El método de la burbuja es una comparación lineal con cada uno de los elementos, el elemento que sea menor contra el que se está comparado intercambiaran posiciones. Este método no es recomendado para grandes comparaciones, ya que es un proceso muy lento y requiere de una gran cantidad de Memoria RAM. Este proceso deberá repetirse recorriendo todo el arreglo hasta que no ocurra ningún intercambio. (Navarro, 2016)

- Doble burbuja con señal:

  Se basa en el algoritmo de burbuja, pero con la adición de una señal que indica si se ha producido algún intercambio durante una pasada completa por la lista. Si no se ha producido ningún intercambio, se considera que la lista está ordenada y se finaliza el proceso.

  Este algoritmo puede reducir el número de comparaciones y, por lo tanto, mejorar la eficiencia en algunos casos en comparación con el algoritmo de burbuja estándar. Pero sigue teniendo una complejidad de tiempo, por lo que puede no ser eficiente para conjuntos de datos grandes. (Navarro, 2016)

- Shell (incrementos – decrementos):

  Se basa en la idea de dividir la lista en subgrupos más pequeños y ordenarlos de forma independiente utilizando un intervalo o incremento específico. Para después reducir el tamaño del incremento hasta que finalmente se realiza una

pasada completa por la lista para realizar los últimos ajustes y asegurar que todos los elementos estén en su posición correcta.

Funciona comparando elementos que estén distantes; la distancia entre comparaciones decrece conforme el algoritmo se ejecuta hasta la última fase, en la cual se comparan los elementos adyacentes, por esta razón se le llama ordenación por disminución de incrementos. Permite intercambios de elementos que están separados por una distancia mayor que 1. Al dividir la lista en subgrupos más pequeños y ordenarlos de forma independiente, se logra un efecto de "pre-ordenamiento" que facilita el proceso final de ordenación.

El algoritmo de Shell puede ser más eficiente que otros algoritmos de ordenamiento, especialmente cuando se trabaja con listas de tamaño mediano, pero su desempeño varía dependiendo del intervalo o incremento elegido, y no siempre garantiza el ordenamiento óptimo en todos los casos.

- Selección directa:

Consiste en buscar el menor elemento del arreglo y colocarlo en la primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se coloca en la segunda posición. El proceso continuo hasta que todos los elementos del arreglo han sido ordenados. Es un algoritmo simple y eficiente pero puede necesitar un alto número de intercambios, lo que puede afectar su rendimiento en listas grandes. Aun así, es adecuado para listas de tamaño moderado y es fácil de implementar.

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

- Inserción directa:

Consiste en recorrer la lista de izquierda a derecha, tomando cada elemento y colocándolo en su posición correcta dentro de la porción ordenada de la lista.

En cada iteración, se toma un elemento de la lista no ordenada y se compara con los elementos previos en la porción ordenada. Si el elemento es menor, se desplazan los elementos mayores a la derecha para abrir espacio y se inserta el elemento en la posición adecuada. Este proceso se repite hasta que todos los elementos estén en su posición correcta.

Es eficiente en listas pequeñas o casi ordenadas, pero puede volverse menos eficiente en listas desordenadas o grandes.

- Binaria:

Utiliza una estrategia de búsqueda binaria para insertar cada elemento en su posición correcta dentro de una porción ordenada de la lista. Toma un elemento

de la lista no ordenada y se realiza una búsqueda binaria en la porción ordenada para encontrar la posición en la que debe insertarse. La búsqueda binaria compara el elemento con el valor central de la porción ordenada y determina si debe colocarse antes o después de dicho valor. Luego, se ajusta el rango de búsqueda y se repite el proceso hasta encontrar la posición correcta. Una vez encontrada, se inserta el elemento en esa posición y se actualiza la porción ordenada de la lista. Esto ayuda a acelerar el algoritmo de ordenación reduciendo el número de comparaciones necesarias.

Es eficiente en listas grandes debido a la reducción de comparaciones, pero puede requerir más tiempo de implementación. (Pestaña Jiménez, 2015)

- HeapSort:

Es un método de ordenamiento basado en estructuras de datos llamadas "heaps" o montículos. Utiliza una estructura de datos llamada heap para ordenar los elementos. Se construye un heap a partir de la lista inicial, se extrae el elemento raíz para obtener la lista ordenada y se repite el proceso hasta que todos los elementos hayan sido extraídos. Es un algoritmo eficiente en términos de tiempo, pero puede requerir más espacio de memoria.

Este método es más lento que otros, pero es más eficaz en escenarios más rigurosos. (Pestaña Jiménez, 2015)

- Quicksort recursivo:

Es un método de ordenamiento que se basa en el principio de división y conquista. Se elige un elemento pivote de la lista y se reorganizan los elementos de manera que los elementos más pequeños que el pivote estén a su izquierda, y los elementos más grandes estén a su derecha. Luego, se aplica recursivamente el mismo proceso a las dos sublistas resultantes, hasta que toda la lista esté ordenada. (Jindal, DelftStack, 2023)

Normalmente se toma como pivote el primer elemento de arreglo, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.

Es eficiente en la mayoría de los casos, pero su rendimiento puede verse afectado en casos donde la lista está parcialmente ordenada o contiene elementos duplicados, ya que puede llevar a una mala elección del pivote y aumentar la complejidad del algoritmo. (Jindal, DelftStack, 2021)

- Radix:

Es un método que se basa en los dígitos de los números para organizarlos en forma ascendente o descendente. En lugar de comparar los elementos

directamente, se realiza una clasificación en base a los dígitos de cada elemento. Comienza clasificando los elementos según el valor del dígito menos significativo. Luego, se realiza una clasificación en función del siguiente dígito y se repite este proceso hasta que se hayan clasificado todos los dígitos. Con cada clasificación, los elementos se reorganizan en nuevas "cajas" o contenedores, hasta que finalmente se obtiene la lista ordenada completa.

Además, no se requiere realizar comparaciones entre los elementos, lo que puede resultar en un mejor rendimiento pero puede tener un rendimiento variable en listas con diferentes longitudes de números. (Fernández, 2019)

5.2 Algoritmos de Ordenamiento Externos.

El ordenamiento externo es utilizado cuando la cantidad de datos a ordenar excede la capacidad de la memoria principal de una computadora, por lo que se requiere utilizar un tipo de memoria más lento, como un disco duro. En este caso, los datos a ordenar se encuentran almacenados en archivos en un dispositivo de almacenamiento externo.

El proceso de ordenamiento externo implica dividir los datos en bloques más pequeños que puedan ser manejados por la memoria principal, y luego realizar una serie de operaciones de ordenamiento en estos bloques. Para después realizar una mezcla de los bloques ordenados para obtener el archivo finalmente ordenado.

El ordenamiento externo es particularmente útil cuando se trabaja con conjuntos de datos muy grandes que no caben en la memoria principal. Aunque es más lento que el ordenamiento interno debido a los tiempos de acceso al dispositivo de almacenamiento externo, el ordenamiento externo permite manejar y ordenar grandes volúmenes de datos de manera efectiva.

El objetivo principal del ordenamiento externo es minimizar la cantidad de accesos al dispositivo de almacenamiento externo, ya que es mucho más lento en comparación con la memoria principal. El ordenamiento externo es esencial en situaciones donde la cantidad de datos a ordenar supera la capacidad de la memoria principal, como la clasificación de grandes volúmenes de datos en bases de datos, procesamiento de registros de transacciones o la ordenación de archivos de registro.

5.2.1 Intercalación:

En este método existen dos archivos con llaves ordenadas, los cuales se mezclan para formar un solo archivo. El proceso consiste en leer un registro de cada archivo y compararlos, el menor es almacenando en el archivo de resultado y el otro se compara con el siguiente elemento del archivo si existe. El proceso se repite hasta que alguno de los archivos quede vacío y los elementos del otro archivo se almacenan directamente en el archivo resultado.

Permite ordenar conjuntos de datos que superan la capacidad de la memoria principal, ya que solo se requiere cargar y procesar bloques más pequeños a la vez. Lo malo es que puede requerir más operaciones de lectura y escritura en disco, lo que puede hacer que el proceso de ordenamiento sea más lento en comparación con los algoritmos de ordenamiento internos.

5.2.2 Mezcla Directa:

En este método, los datos se dividen en bloques más pequeños que pueden ser manejados por la memoria principal, y se realiza una mezcla directa de los bloques para obtener la lista final ordenada.

comienza dividiendo los datos en bloques que puedan ser cargados en la memoria principal. Cada bloque se carga en memoria y se ordena internamente utilizando un algoritmo de ordenamiento, como Quicksort. Luego, se realiza una mezcla directa de los bloques ordenados, donde se toman los elementos más pequeños de cada bloque y se colocan en una lista de salida.

Durante el proceso de mezcla directa, los bloques se van leyendo y escribiendo en disco según sea necesario, evitando cargar todos los datos en memoria al mismo tiempo. Se selecciona el elemento más pequeño de cada bloque y se agrega a la lista de salida, y luego se lee el siguiente elemento del bloque correspondiente para continuar con la mezcla. La mezcla directa continúa hasta que se hayan agotado todos los elementos de los bloques y se haya construido la lista ordenada completa.

Permite ordenar conjuntos de datos que exceden la capacidad de la memoria principal, aunque puede ser más lento debido a las operaciones de lectura y escritura en disco. (Callejas, 2014)

5.2.3 Mezcla Natural:

Este algoritmo se basa en la idea de aprovechar las secuencias ya ordenadas presentes en los datos y combinarlas de manera eficiente. Se comienza dividiendo los datos en secuencias ordenadas de forma ascendente, se realiza una mezcla de estas secuencias, tomando los elementos más pequeños de cada secuencia y colocándolos en una lista de salida. A medida que se van extrayendo los elementos más pequeños, se van leyendo nuevos elementos de las secuencias correspondientes para continuar con la mezcla. Aprovecha las secuencias ya ordenadas presentes en los datos, evitando operaciones innecesarias de comparación. El proceso de mezcla natural continúa hasta que todas las secuencias hayan sido fusionadas y se haya obtenido la lista finalmente ordenada.

Aprovecha las secuencias ordenadas presentes en los datos para realizar una mezcla eficiente. Se identifican y fusionan secuencias de manera dinámica, reduciendo la cantidad de accesos al dispositivo de almacenamiento externo y mejorando el rendimiento del algoritmo. Es eficiente en casos de datos con secuencias ordenadas o patrones de ordenamiento natural.

Burbuja Señal

El método de ordenamiento burbuja es un algoritmo simple que funciona comparando pares de elementos adyacentes y intercambiándolos si están en el orden incorrecto. Este proceso se repite hasta que la lista esté completamente ordenada.
Después de que el algoritmo de ordenamiento burbuja haya terminado, el arreglo estará ordenado de forma ascendente.

Es importante tener en cuenta que el método de ordenamiento burbuja no es eficiente para grandes conjuntos de datos, ya que tiene una complejidad de tiempo cuadrática (O(n^2)). Sin embargo, es útil para enseñar conceptos básicos de algoritmos de ordenamiento.

- ✓ Analisis de eficiencia.

El método de ordenamiento burbuja tiene una complejidad de tiempo cuadrática, lo que significa que su tiempo de ejecución crece proporcionalmente al cuadrado del número de elementos a ordenar. Más precisamente, tiene una complejidad de tiempo promedio y peor caso de O(n^2), donde "n" es el número de elementos en la lista.

Esto se debe a que el algoritmo compara y realiza intercambios entre pares de elementos adyacentes en cada iteración. En el peor caso, cuando la lista está ordenada en orden descendente, se deben realizar intercambios en cada comparación para llevar el elemento más grande hasta la última posición. Esto implica que se necesitan "n" iteraciones para colocar el elemento más grande en su posición final, y se repite este proceso para cada elemento restante en la lista, dando como resultado "n" iteraciones adicionales. En total, se realizan aproximadamente n * n = n^2 comparaciones.

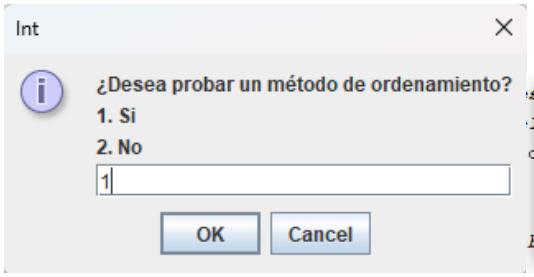
Además, el método de ordenamiento burbuja requiere que se realicen múltiples pasadas completas por la lista para asegurarse de que todos los elementos estén en su posición correcta. En cada pasada, el elemento más grande "burbujea" hacia su posición final. Por lo tanto, el número total de pasadas necesarias es igual al número de elementos en la lista, es decir, "n".

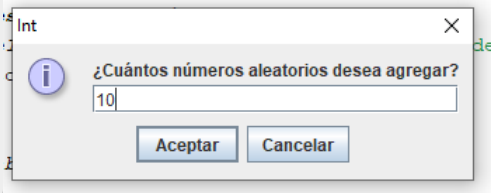
En términos de espacio, el algoritmo de ordenamiento burbuja es in situ (in-place), lo que significa que no requiere memoria adicional aparte de la lista original. Los intercambios se realizan directamente en el arreglo dado, sin utilizar estructuras de datos adicionales.

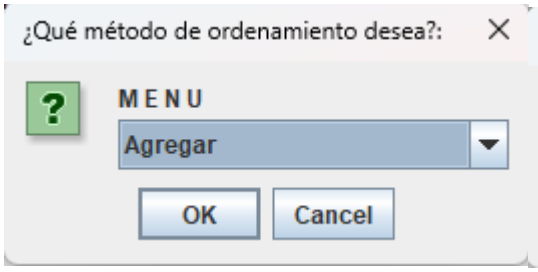
En resumen, el método de ordenamiento burbuja es fácil de implementar, pero no es eficiente para grandes conjuntos de datos debido a su complejidad de tiempo cuadrática.
Hay algoritmos más eficientes, como el ordenamiento por mezcla (merge sort) o el

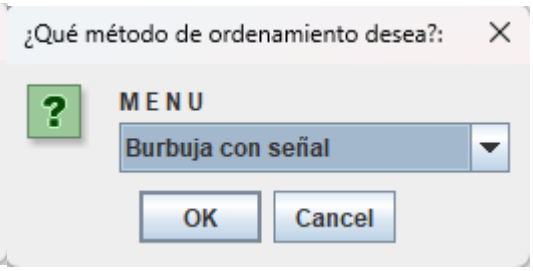
ordenamiento rápido (quick sort), que tienen una complejidad de tiempo inferior a O(n^2) y son preferibles cuando se trabaja con grandes cantidades de datos.

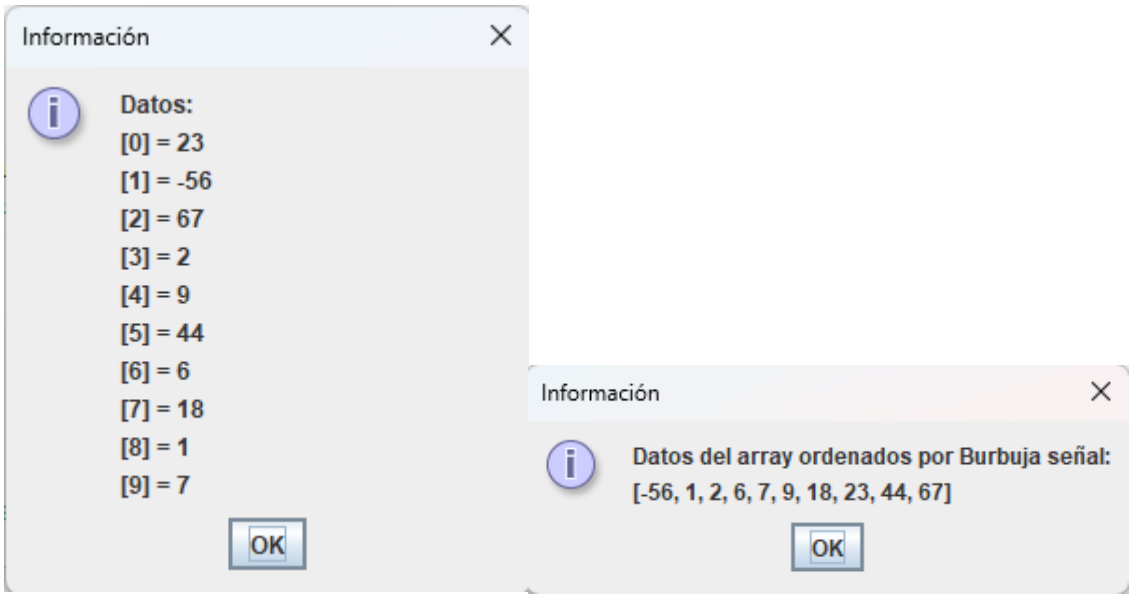
- ✓ Análisis de los casos.
  1. Mejor caso: El mejor caso ocurre cuando la lista ya está completamente ordenada. En este escenario, el algoritmo de ordenamiento burbuja realizará una pasada completa por la lista para verificar que no haya intercambios, pero no realizará ningún intercambio adicional. En términos de complejidad de tiempo, esto significa que se necesitarán n - 1 comparaciones en el bucle externo y ninguna en el bucle interno. Por lo tanto, la complejidad de tiempo en el mejor caso es O(n).
  2. Caso medio: En el caso medio, el comportamiento del algoritmo de ordenamiento burbuja depende de la distribución de los elementos en la lista. En promedio, se espera que se realicen alrededor de n^2/4 comparaciones e intercambios. Por lo tanto, la complejidad de tiempo en el caso medio es O(n^2).
  3. Peor caso: El peor caso ocurre cuando la lista está ordenada en orden descendente, lo que implica que se deben realizar intercambios en cada comparación para llevar el elemento más grande hasta la última posición. En este escenario, se realizarán n - 1 comparaciones en la primera pasada, n - 2 comparaciones en la segunda pasada, y así sucesivamente, hasta realizar 1 comparación en la última pasada. En total, se realizarán aproximadamente n * (n - 1) / 2 comparaciones, lo cual es equivalente a una complejidad de tiempo de O(n^2) en el peor caso.
- ✓ Prueba de escritorio.











- **Doble Burbuja.**

El método de la doble burbuja, también conocido como "Bubble Sort" en inglés, es un algoritmo de ordenamiento simple que funciona comparando repetidamente pares de elementos adyacentes y intercambiándolos si están en el orden incorrecto. El proceso se repite hasta que la lista esté ordenada.

- ✓ Analisis de eficiencia

El método de la doble burbuja tiene una complejidad de tiempo promedio de O(n^2), donde 'n' es el número de elementos en la lista a ordenar. Esto significa que el tiempo de ejecución del algoritmo crece cuadráticamente con el tamaño de la lista.

La doble burbuja es un algoritmo sencillo de implementar, pero no es eficiente para grandes conjuntos de datos debido a su alta complejidad cuadrática. En comparación con otros algoritmos de ordenamiento más eficientes, como QuickSort o MergeSort, que tienen una complejidad promedio de O(n log n), la doble burbuja tiende a ser mucho más lenta.

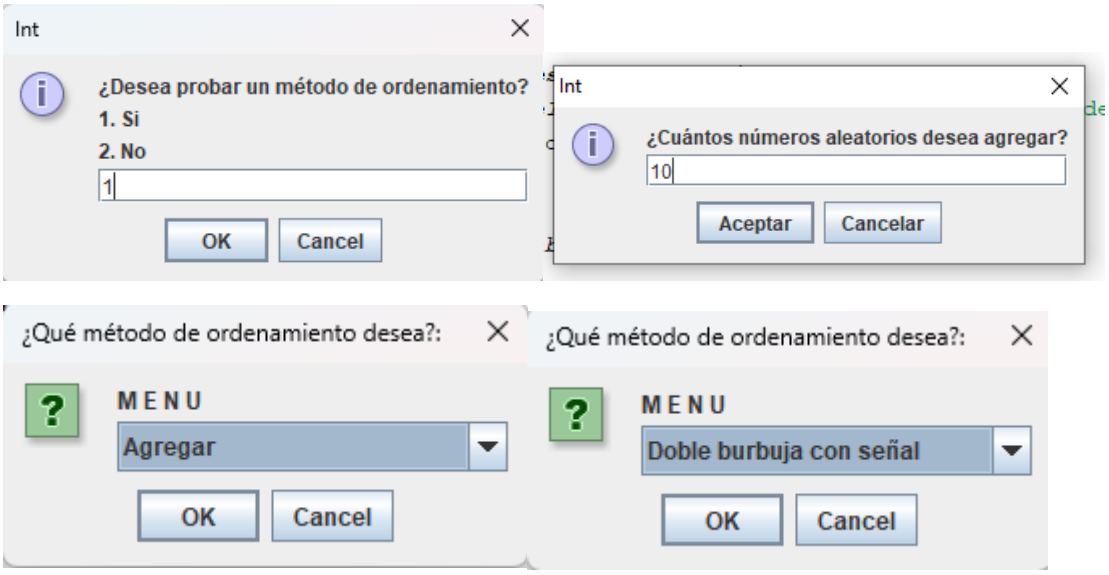
Sin embargo, el método de la doble burbuja puede ser útil en ciertos casos específicos, como cuando se tiene una lista pequeña o cuando la lista está casi ordenada. En estos casos, el algoritmo puede tener un rendimiento aceptable.

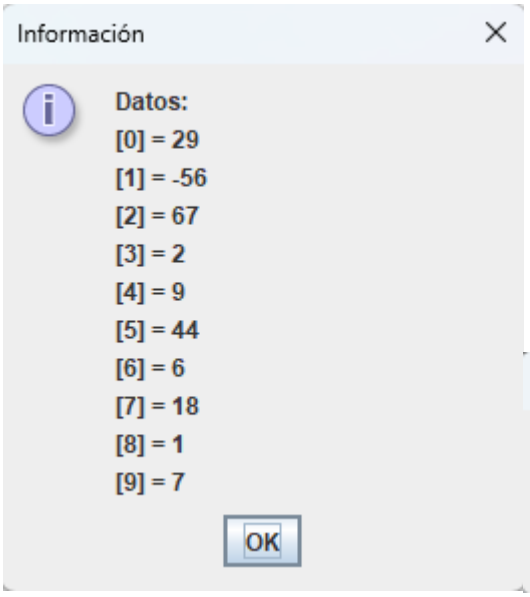
- ✓ Análisis de los casos.
  1. Mejor caso: El mejor caso ocurre cuando la lista ya está completamente ordenada. En este escenario, el algoritmo de la doble burbuja aún realiza comparaciones en cada par de elementos, pero no se realizan intercambios, ya que todos los elementos ya están en su posición correcta. Esto significa que el bucle interno se ejecutará n-1 veces en cada iteración del bucle externo. La complejidad de tiempo en el mejor caso sigue siendo cuadrática, O(n^2), pero se reduce en comparación con

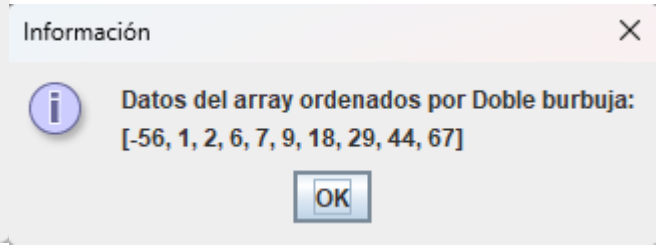
otros casos. Sin embargo, este caso no mejora significativamente la eficiencia del algoritmo y sigue siendo relativamente lento.

2. Caso medio: En el caso medio, el ordenamiento de la lista requerirá realizar intercambios en aproximadamente la mitad de las comparaciones. La cantidad total de comparaciones e intercambios realizados en el bucle interno seguirá siendo n*(n-1)/2, lo cual es igual a O(n^2). Aunque se espera que el algoritmo sea más eficiente que el peor caso, la complejidad cuadrática sigue siendo un factor limitante. En general, el caso medio del método de la doble burbuja es similar al peor caso en términos de eficiencia.
3. Peor caso: El peor caso ocurre cuando la lista está en orden descendente o en un orden inverso al orden deseado. En este escenario, se requieren intercambios en todas las comparaciones del bucle interno. El algoritmo realizará aproximadamente n*(n-1)/2 comparaciones e intercambios en total, lo cual es igual a O(n^2). Esto implica que el algoritmo de la doble burbuja es muy ineficiente para conjuntos de datos grandes en el peor caso. A medida que el tamaño de la lista aumenta, el tiempo de ejecución del algoritmo se incrementa significativamente.

✓ Prueba de escritorio.







- **Shell (incrementos – decrementos)**

Aquí nos basamos en comparar e intercambiar los elementos, esto por medio de usar incrementos decrecientes para dividir el conjunto de los datos que tenemos en subconjuntos, a cada uno de ellos se aplicará el algoritmo de inserción directa. A medida que los incrementos disminuyen se va logrando un mejor orden de todo el conjunto.

- ✓ Análisis de eficiencia.

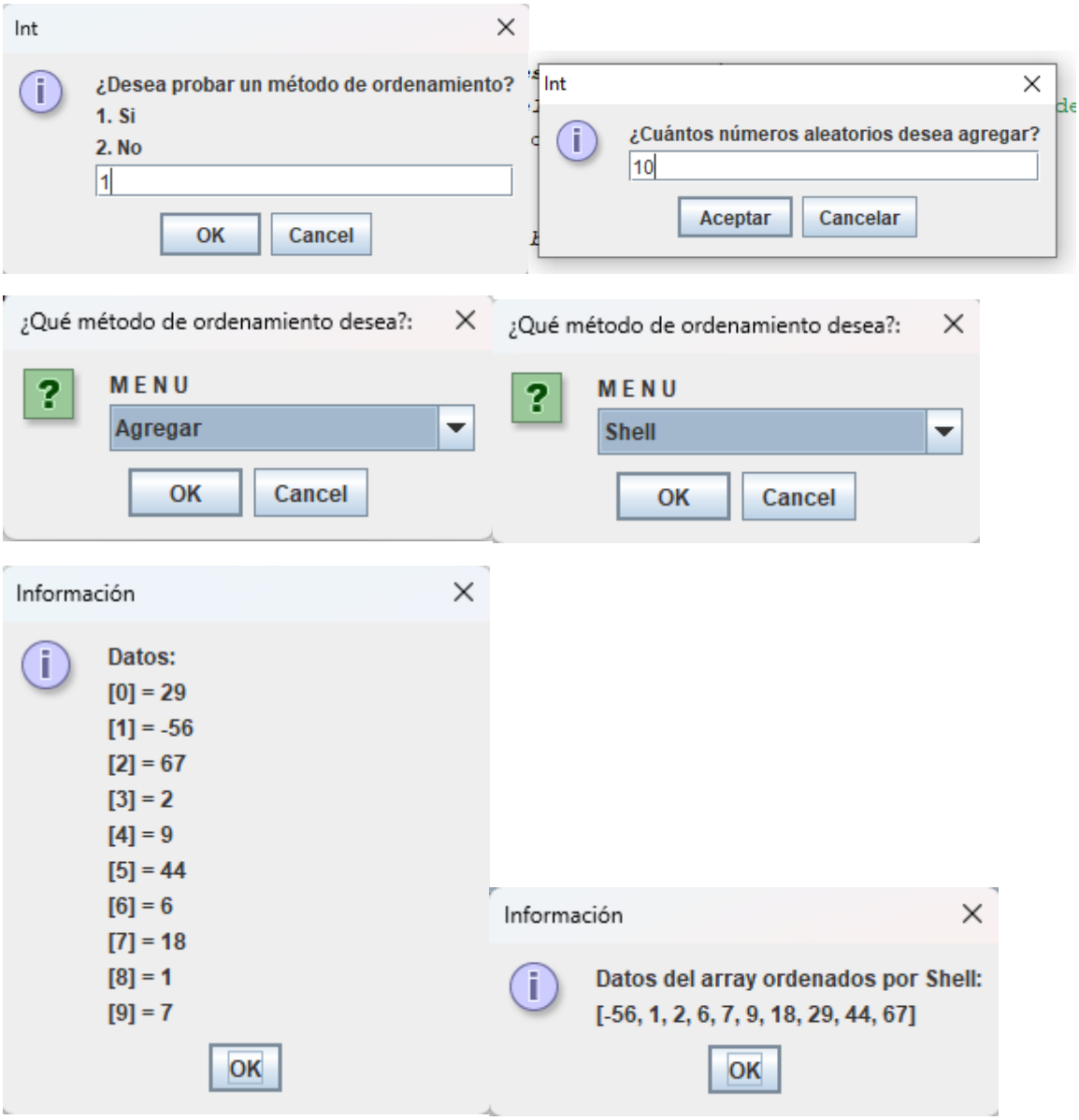
Es un método dependiente de los incrementos utilizados; su complejidad temporal se sitúa entre O(n log n) y O(n^2), dependiendo de la secuencia de incrementos utilizada. En términos de complejidad espacial, ShellSort también es eficiente, ya que solo requiere una cantidad constante de espacio adicional para realizar las operaciones de ordenamiento.

- ✓ Análisis de los casos.

1. Mejor caso: En el mejor caso, ShellSort tiene una complejidad temporal cercana a O(n log n). Esto ocurre cuando la secuencia de incrementos utilizada permite que los elementos ya estén casi ordenados en su posición final. En este caso, el algoritmo realiza una **cantidad mínima de comparaciones y desplazamientos**, mejorando su eficiencia.
2. Caso medio: En el caso medio, ShellSort tiene una complejidad temporal que varía entre O(n log n) y O(n^2), dependiendo de la secuencia de incrementos utilizada. Este caso ocurre cuando los elementos del conjunto de datos están desordenados de forma aleatoria. El algoritmo ShellSort **sigue dividiendo el conjunto en subconjuntos más pequeños y aplicando el algoritmo de inserción directa a cada uno de ellos, mejorando gradualmente la ordenación del conjunto completo.**

3. Peor caso: El peor caso de ShellSort ocurre cuando la secuencia de incrementos utilizada no es eficiente para la ordenación del conjunto de datos. En este caso, la complejidad temporal puede alcanzar hasta O(n^2), lo que implica un **mayor número de comparaciones y desplazamientos**. Sin embargo, incluso en el peor caso, ShellSort tiende a ser más eficiente que otros algoritmos cuadráticos como la selección directa o la inserción directa.

✓ Prueba de escritorio.



Selección directa.

Se basa en la búsqueda del elemento mínimo o máximo en cada pasada y su intercambio con el elemento correspondiente. A medida que se realizan las pasadas, los elementos se van ubicando en su posición final.

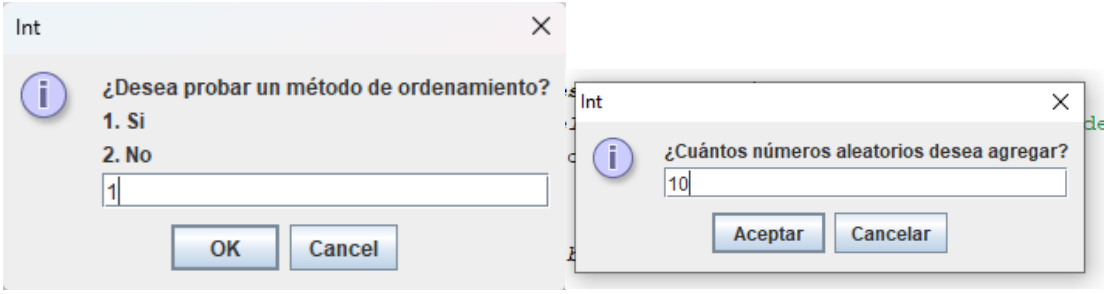
- ✓ Análisis de eficiencia.

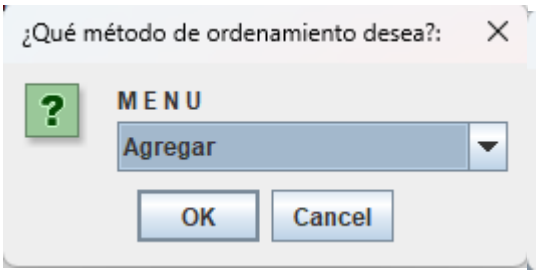
En cada recorrido se deben realizar comparaciones y encontrar el elemento mínimo o máximo. En términos de complejidad espacial, la selección directa es eficiente, ya que solo requiere una cantidad constante de espacio adicional para realizar las operaciones de ordenamiento.

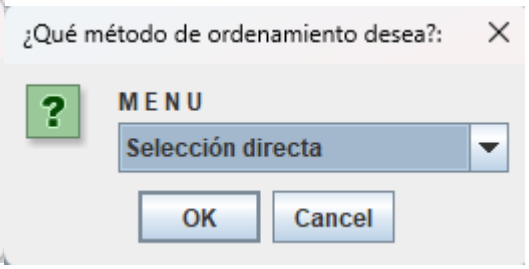
- ✓ Análisis de los casos.

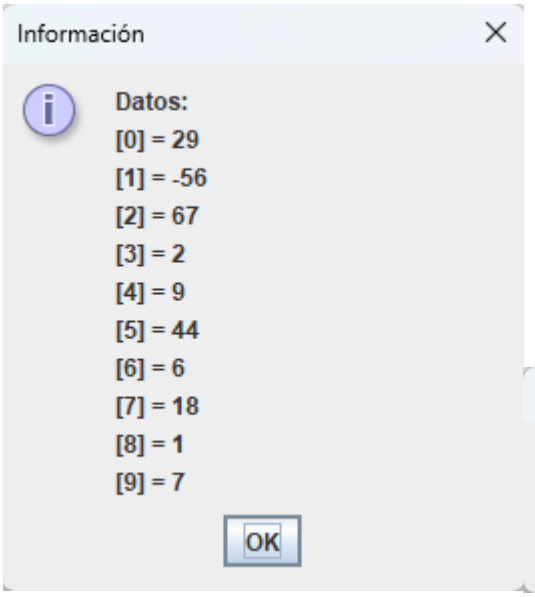
1. Mejor caso: En el mejor caso, la selección directa tiene una complejidad temporal de O(n^2). Esto ocurre cuando **el conjunto de datos ya está ordenado o casi ordenado.** Aunque en este caso se realizan menos comparaciones, aún es necesario recorrer el conjunto para encontrar el elemento mínimo o máximo en cada pasada.
2. Caso medio: En el caso medio, la selección directa también tiene una complejidad temporal de O(n^2). Este caso ocurre cuando los elementos del conjunto de datos están desordenados de forma aleatoria. El algoritmo sigue realizando comparaciones en cada pasada para encontrar el elemento mínimo o máximo, lo que implica una **cantidad significativa de operaciones**.
3. Peor caso: El peor caso de la selección directa ocurre cuando **el conjunto de datos está ordenado en forma descendente**. En este caso, se deben realizar el máximo número de comparaciones y movimientos de elementos para lograr la ordenación correcta. La complejidad temporal sigue siendo de O(n^2), lo que lo hace menos eficiente que otros algoritmos de ordenamiento.

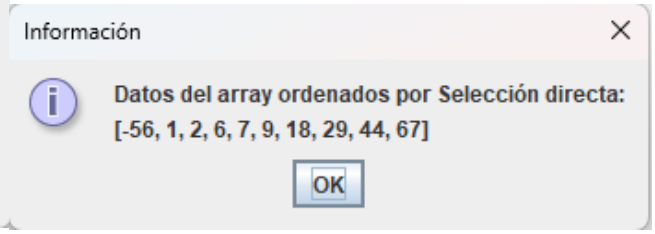
- ✓ Prueba de escritorio.











Inserción directa.

Se basa en la inserción secuencial de elementos en una lista ordenada. A medida que se inserta cada elemento, se ajusta su posición para mantener el orden.

- ✓ Análisis de eficiencia.

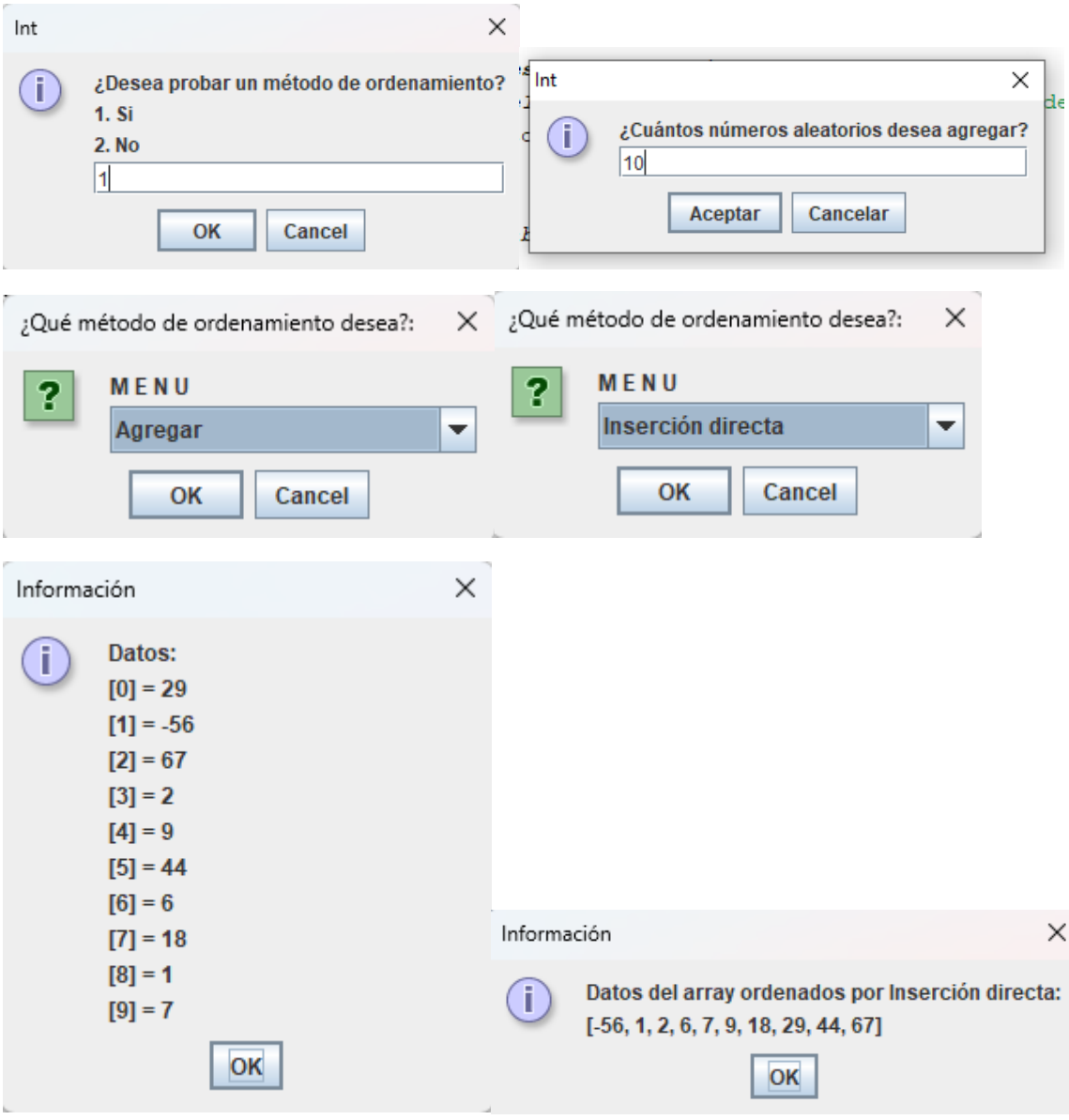
La inserción directa tiene una complejidad temporal de O(n^2) en todos los casos. En cada pasada, se deben realizar comparaciones y desplazamientos de elementos para encontrar la posición correcta de inserción. En términos de complejidad espacial, la inserción directa es eficiente, ya que solo requiere una cantidad constante de espacio adicional para realizar las operaciones de ordenamiento.

- ✓ Análisis de los casos.

1. Mejor caso: En el mejor caso, la complejidad temporal de la inserción directa es de O(n). En este caso, se necesitará realizar un **número mínimo de comparaciones y desplazamientos**, ya que la mayoría de los elementos ya están en su posición correcta.

2. Caso medio: En el caso medio, la inserción directa tiene una complejidad temporal de O(n^2). Este caso ocurre cuando los **elementos del conjunto de datos están desordenados de forma aleatoria**.
3. Peor caso: En este caso, se deben realizar el **máximo número de comparaciones y desplazamientos para lograr la ordenación correcta**. La complejidad temporal sigue siendo de O(n^2), lo que la hace menos eficiente en comparación con otros algoritmos de ordenamiento en este caso.

✓ Prueba de escritorio.



Binaria

✓ Análisis de eficiencia:

un método de ordenamiento binario puede ofrecer una eficiencia promedio de O(n log n), pero puede tener un peor caso de O(n^2). Es importante tener en cuenta que existen otros métodos de ordenamiento más eficientes en términos de tiempo, como Quicksort o Mergesort. La elección del método de ordenamiento adecuado dependerá de los requisitos específicos del problema y del tamaño del conjunto de datos a ordenar.

✓ Análisis de los casos:

Un método de ordenamiento binario, en el contexto de esta conversación, se refiere a un algoritmo de ordenamiento que utiliza la representación binaria de los elementos para realizar las comparaciones y los intercambios. A continuación, se presenta un análisis de los diferentes casos que pueden presentarse en un método de ordenamiento binario:

1. Mejor caso:

Descripción: El mejor caso ocurre cuando el arreglo ya está completamente ordenado.

Comportamiento: En este caso, el algoritmo de ordenamiento binario realizará las comparaciones necesarias para verificar que los elementos están en orden, pero no se realizarán intercambios.

Complejidad temporal: En el mejor caso, la complejidad temporal puede ser O(n), ya que solo se requiere recorrer el arreglo una vez para verificar que está ordenado.

2. Caso promedio:

Descripción: El caso promedio se refiere a una distribución aleatoria de los elementos en el arreglo.

Comportamiento: En este caso, el algoritmo de ordenamiento binario realizará las comparaciones y los intercambios necesarios para ordenar el arreglo utilizando la representación binaria de los elementos.

Complejidad temporal: En el caso promedio, la complejidad temporal esperada es O(n log n), donde "n" es el tamaño del arreglo. Esto se debe a que el algoritmo realiza comparaciones y desplazamientos utilizando la representación binaria de los elementos.

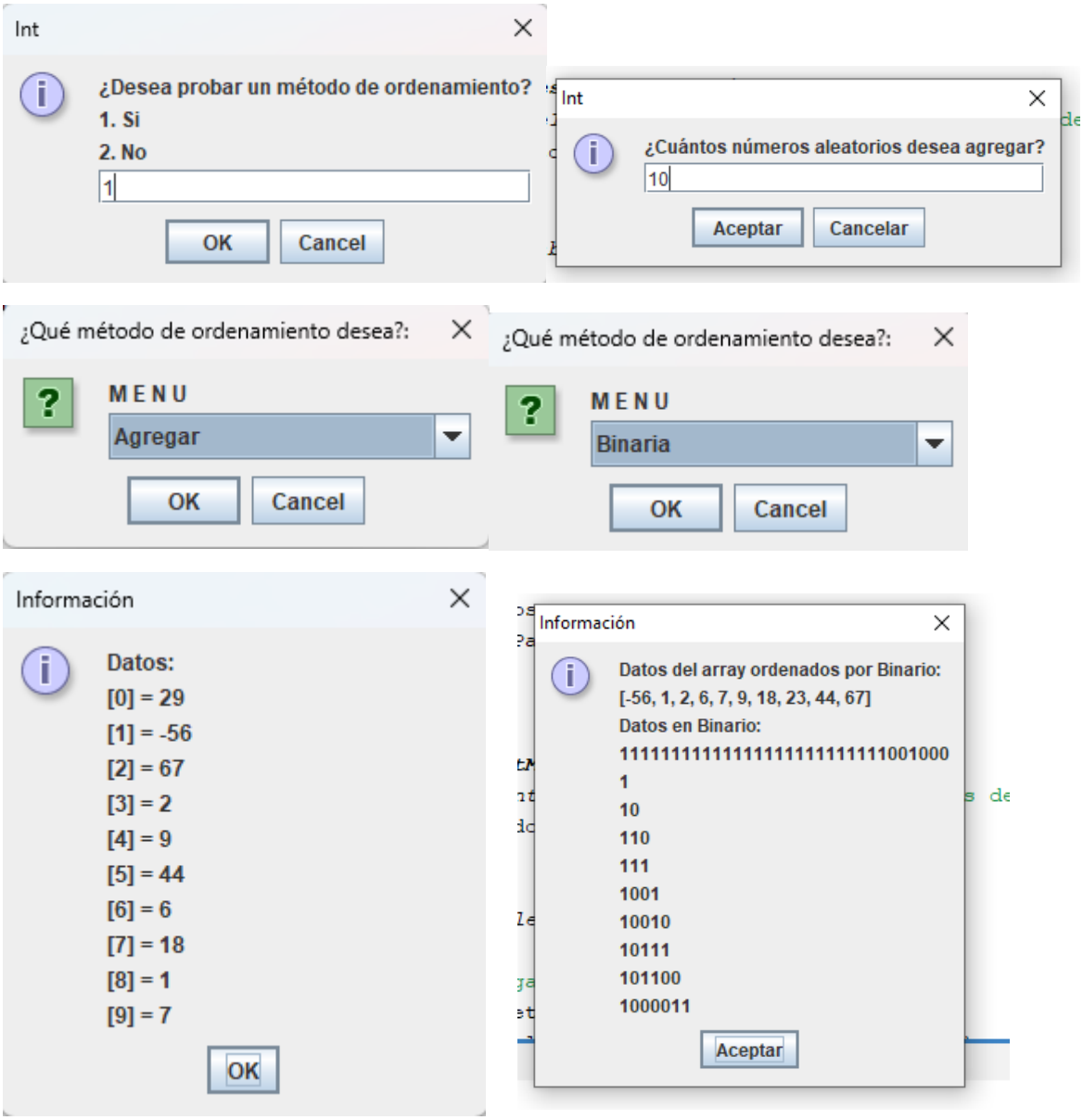
3. Peor caso:

Descripción: El peor caso ocurre cuando los elementos están inicialmente ordenados en orden descendente.

Comportamiento: En este caso, el algoritmo de ordenamiento binario realizará múltiples pasadas de comparaciones y desplazamientos para ordenar los elementos en orden ascendente.

Complejidad temporal: En el peor caso, la complejidad temporal puede ser O(n^2), donde "n" es el tamaño del arreglo. Esto ocurre debido a la cantidad de comparaciones y desplazamientos necesarios para ordenar los elementos en orden ascendente.

✓ Prueba de escritorio:



Heap Sort.

Heap Sort es un algoritmo de ordenamiento interno basado en la estructura de datos conocida como "heap" o montículo. Se caracteriza por su eficiencia y capacidad para ordenar conjuntos de datos de manera óptima.

- ✓ Análisis de eficiencia:

Es conocido por su eficiencia y capacidad para ordenar conjuntos de datos de manera óptima. Su análisis de eficiencia se basa en el uso de una estructura de datos llamada "heap" (montículo), que permite una rápida selección y extracción del elemento máximo o mínimo.

En términos de complejidad espacial, es muy eficiente, ya que solo requiere una cantidad constante de espacio adicional para mantener el heap y realizar las operaciones de ordenamiento en su lugar. Esto significa que el algoritmo no necesita asignar memoria adicional proporcional al tamaño del conjunto de datos a ordenar, lo que lo hace muy adecuado para conjuntos de datos grandes.

- ✓ Análisis de los casos:

1. Mejor de los casos:

Tiene una complejidad temporal de O(n log n). Ocurre cuando el conjunto de datos ya está ordenado en forma ascendente o descendente. El proceso de construcción del heap y la restauración del heap después de cada extracción no requieren muchos intercambios, ya que los elementos ya están en la posición correcta.

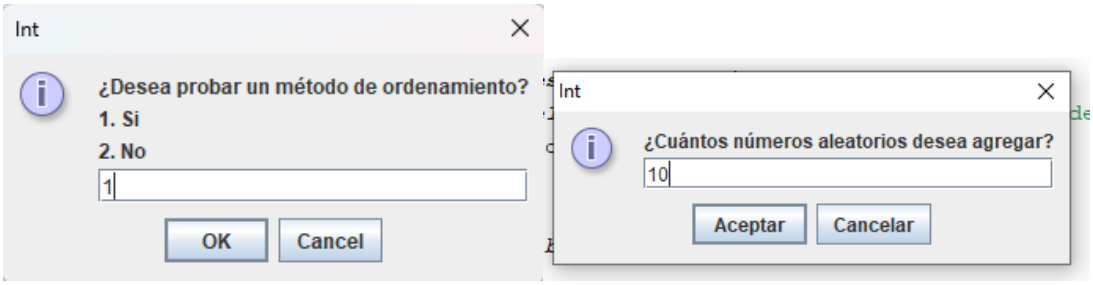
2. Caso medio:

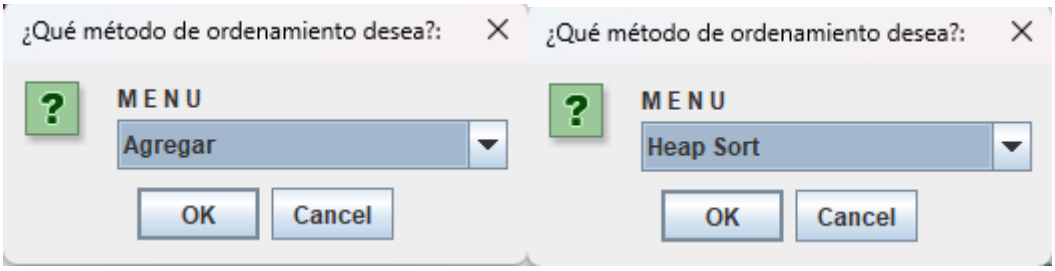
Tiene una complejidad temporal de O(n log n). Ocurre cuando el conjunto de datos está desordenado de forma aleatoria. El HeapSort sigue construyendo el heap y realizando las operaciones de extracción y restauración del heap. Aunque la construcción inicial del heap puede requerir una cantidad significativa de tiempo, el proceso de extracción y restauración se repite de manera eficiente y logra ordenar el conjunto de datos en el tiempo esperado.

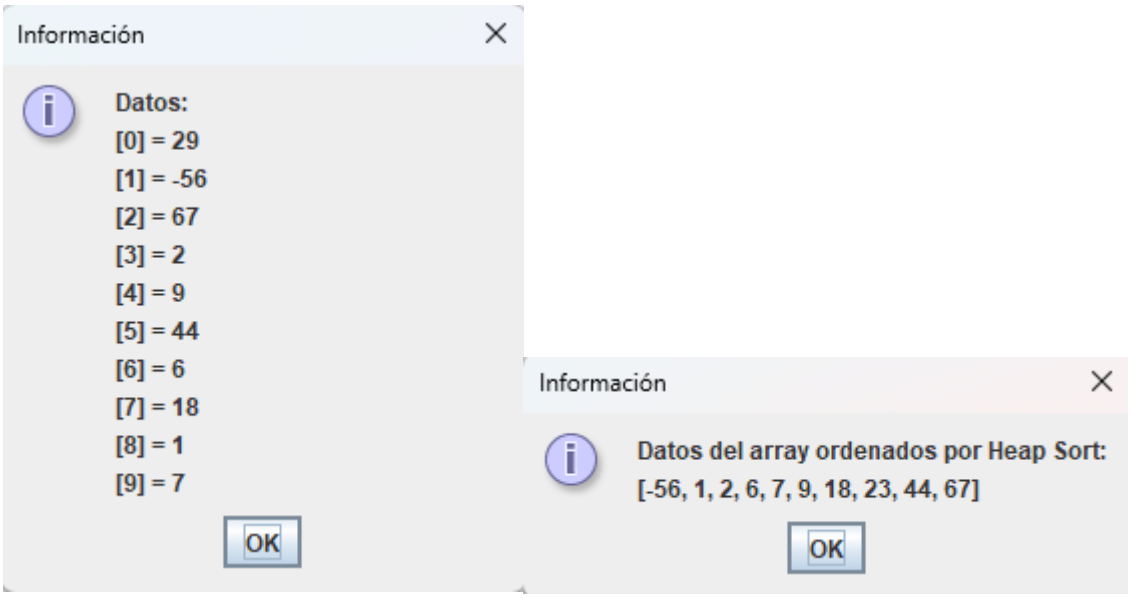
3. Peor caso:

Ocurre cuando el conjunto de datos está ordenado de forma descendente. En este escenario, cada elemento a insertar en el heap debe descender hasta su posición adecuada, lo que implica una mayor cantidad de intercambios y operaciones de restauración del heap. Aunque HeapSort sigue siendo eficiente en el peor caso, puede tener una constante oculta más alta en comparación con otros algoritmos de ordenamiento.

- ✓ Prueba de escritorio:







Quick sort recursivo

Este método funciona de la siguiente forma: Se verifica si el arreglo es nulo o está vacío. En ese caso, no se hace nada, ya que un arreglo vacío o con un solo elemento ya se considera ordenado.

Se selecciona un pivote como referencia. En este caso, se selecciona el elemento central del arreglo.

Se divide el arreglo en dos partes, de manera que los elementos menores al pivote se encuentren a su izquierda y los mayores a su derecha.

Se repite el proceso de manera recursiva para las dos sub-partes generadas, hasta que todos los elementos estén ordenados.

Durante la partición, se utilizan dos índices (i y j) para buscar elementos que deban intercambiarse. Cuando se encuentran dichos elementos, se intercambian y se incrementa i y se decrementa j.

El proceso de partición se realiza hasta que i es mayor que j, momento en el cual se ha dividido correctamente el arreglo en dos partes.

Finalmente, se realizan las llamadas recursivas a Quicksort para ordenar las dos sub-partes generadas, siempre y cuando tengan al menos un elemento.

La función swap se utiliza para intercambiar dos elementos en el arreglo.

- ✓ Análisis de eficiencia.

El algoritmo Quicksort tiene un tiempo de ejecución promedio de O(n log n), donde "n" representa el tamaño del arreglo a ordenar.

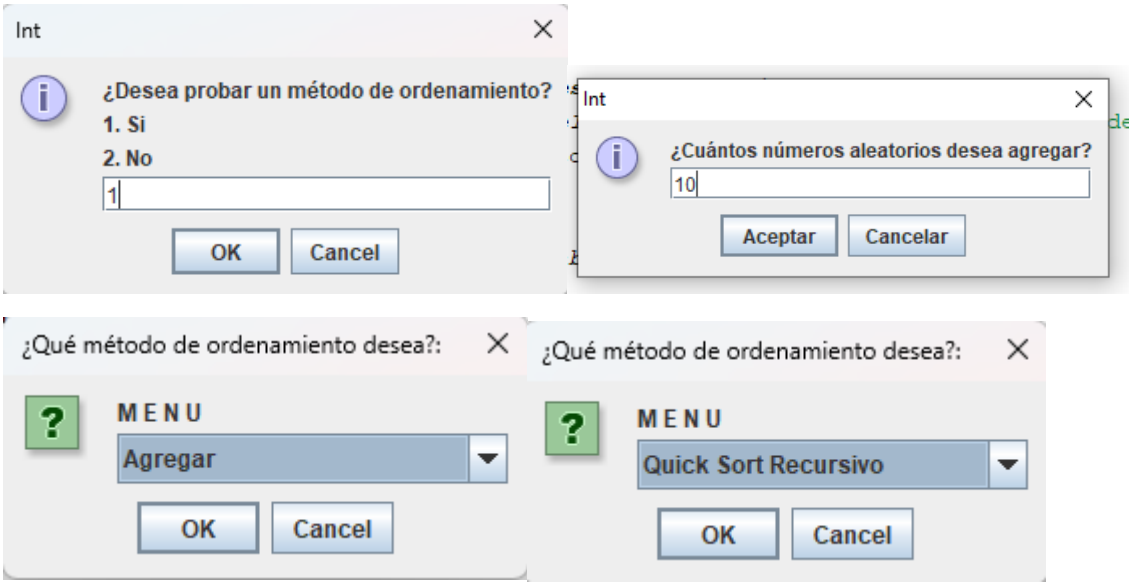
el espacio auxiliar requerido por el algoritmo Quicksort es O(log n) en el caso promedio, debido a las llamadas recursivas que se realizan en la pila de ejecución. En el peor caso, el espacio auxiliar requerido puede llegar a ser O(n) si las llamadas recursivas están profundamente anidadas.

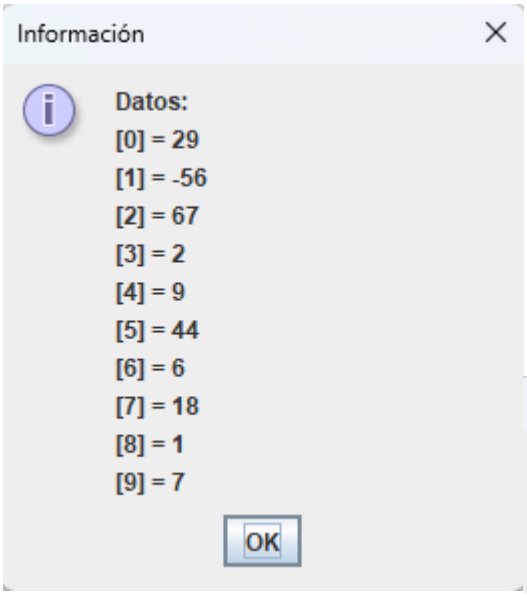
- ✓ Análisis de los casos.
    1. Mejor caso: El mejor caso ocurre cuando en cada iteración del algoritmo, el pivote seleccionado divide el arreglo en dos partes de igual tamaño. En este escenario, el tiempo de ejecución es óptimo y tiene una complejidad de O(n log n).
       En el mejor caso, el algoritmo Quicksort realiza particiones equilibradas y divide el arreglo en dos mitades en cada iteración. Esto se puede lograr seleccionando el pivote como el elemento central del arreglo o utilizando técnicas avanzadas de selección de pivote.
       En este caso, el número total de comparaciones y movimientos de elementos es mínimo, lo que resulta en un tiempo de ejecución eficiente. El algoritmo tiene una complejidad de tiempo de O(n log n) en el mejor caso.
    2. Caso medio: El caso medio se refiere a una distribución aleatoria de elementos en el arreglo. En este escenario, el algoritmo Quicksort tiene un rendimiento promedio y también tiene una complejidad de tiempo de O(n log n).
       El análisis del caso medio es complejo debido a las diversas permutaciones posibles de los elementos del arreglo. Sin embargo,

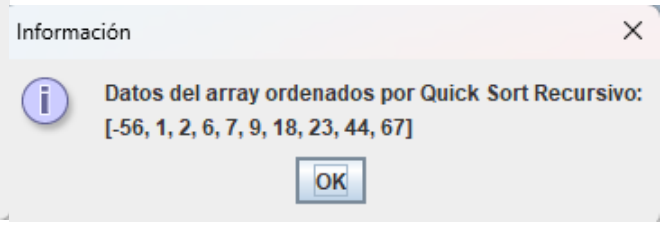
utilizando técnicas de análisis probabilístico, se ha demostrado que el tiempo de ejecución promedio de Quicksort es O(n log n).
En la práctica, el caso medio es el más común y el algoritmo Quicksort se considera eficiente en términos de tiempo de ejecución.

3. Peor caso: El peor caso ocurre cuando el pivote seleccionado en cada iteración divide el arreglo de manera desfavorable, lo que resulta en una partición desequilibrada. Por ejemplo, esto puede ocurrir cuando el arreglo ya está ordenado en orden ascendente o descendente.
   En el peor caso, el tiempo de ejecución de Quicksort es O(n^2), lo cual es menos eficiente en comparación con el mejor y el caso medio. Esto se debe a que el algoritmo realiza muchas más comparaciones y movimientos de elementos en cada iteración.
   Sin embargo, en la práctica, el peor caso ocurre raramente debido a la elección aleatoria del pivote o algoritmos de selección de pivote mejorados, lo que reduce significativamente la probabilidad de que ocurra el peor caso.
   Es importante destacar que la elección del pivote y las técnicas de optimización pueden influir en la eficiencia del algoritmo Quicksort en diferentes casos.

- ✓ Prueba de escritorio.







- **Radix.**

Es un método de ordenamiento no comparativo que se basa en el concepto de clasificar los elementos de una lista según sus dígitos individuales.

El proceso se lleva a cabo en etapas o pasadas, donde en cada pasada se clasifican los elementos según un dígito específico, comenzando por el dígito menos significativo hasta el más significativo. En cada iteración, se utilizan estructuras de datos auxiliares, como colas o listas enlazadas, para agrupar los elementos según el valor del dígito en consideración. El ordenamiento se realiza de manera estable, es decir, se mantiene el orden relativo de los elementos con el mismo valor de dígito.

- ✓ Análisis de eficiencia.

Tiene una eficiencia relativamente buena en términos de tiempo y espacio. Su complejidad temporal es lineal en función del número de dígitos, la cantidad de elementos y el rango de valores. Aunque puede requerir más tiempo y recursos en comparación con algunos otros algoritmos de ordenamiento, este metodo de ordenamiento es especialmente eficiente cuando se aplica a números enteros no negativos con un rango acotado de dígitos.

- ✓ Análisis de los casos.

1. Mejor de los casos:

Tiene una complejidad temporal de O(d * (n + k)), donde ***d*** es el número de dígitos, ***n*** es el número de elementos a ordenar y ***k*** es el rango de valores posibles en los elementos. Este caso ocurre cuando los elementos están uniformemente distribuidos en todos los dígitos y no hay necesidad de reorganización en ninguna pasada.

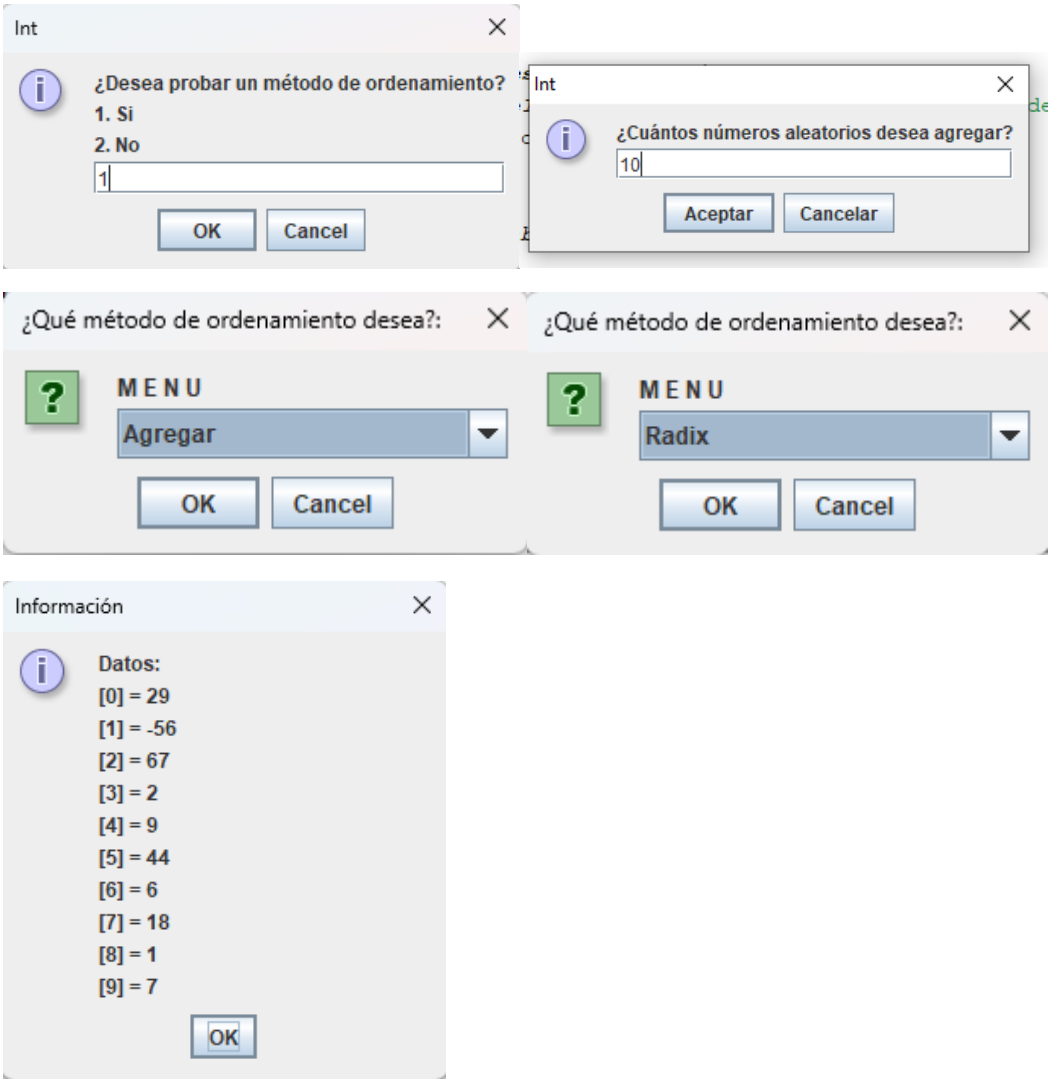
2. Caso medio:

Tiene una complejidad temporal de O(d * (n + k)). Este caso ocurre cuando los elementos están distribuidos de manera más aleatoria en los dígitos. Aunque el rendimiento puede variar dependiendo de la distribución específica de los elementos sigue siendo eficiente en general.

3. Peor caso:

Tiene una complejidad temporal de O(d * (n + k)) y ocurre cuando todos los elementos tienen el mismo valor para todos los dígitos, lo que requiere realizar múltiples pasadas completas para ordenar los elementos. Aunque la complejidad temporal puede ser alta en el peor caso, sigue siendo estable y eficiente en términos generales.

- ✓ Prueba de escritorio.



Intercalación.

✓ Análisis de eficiencia:

el método de ordenamiento por intercalación (merge sort) es eficiente en términos de tiempo, especialmente en el caso promedio, donde su complejidad temporal es de O(n log n). Además, es un algoritmo estable y utiliza una cantidad razonable de memoria adicional. Estas características hacen que el merge sort sea ampliamente utilizado en la práctica para ordenar arreglos de diferentes tamaños.

✓ Análisis de los casos:

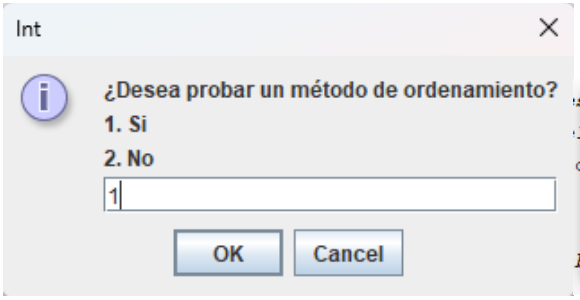
El análisis de casos del método de ordenamiento por intercalación (merge sort) nos permite entender su comportamiento en diferentes escenarios. A continuación, se presenta el análisis de casos del merge sort:

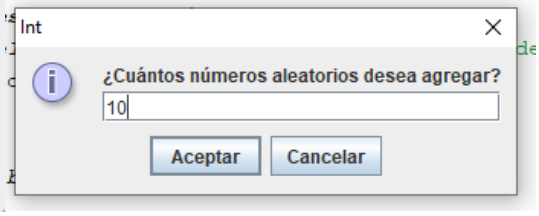
1. Mejor caso: El mejor caso ocurre cuando el arreglo ya está completamente ordenado. En este escenario, el merge sort realiza las divisiones del arreglo original y las fusiones de manera óptima, lo que resulta en un tiempo de ejecución eficiente. En el mejor caso, el merge sort tiene una complejidad temporal de O(n log n), donde "n" es el tamaño del arreglo. Esto se debe a que las divisiones y fusiones del algoritmo son balanceadas y requieren un número mínimo de operaciones.

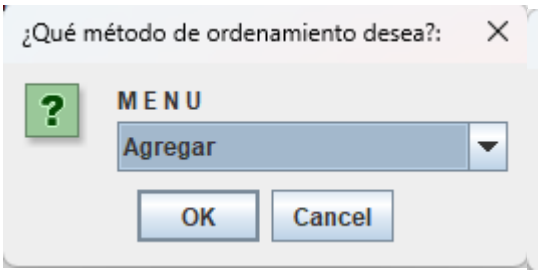
2. Caso promedio: En el caso promedio, el merge sort también tiene una complejidad temporal de O(n log n). Este caso se presenta cuando el arreglo está en un orden aleatorio o desordenado. Aunque puede haber más operaciones de división y fusión en comparación con el mejor caso, el merge sort sigue siendo eficiente y se acerca a su complejidad temporal óptima. El algoritmo divide el arreglo en mitades y fusiona las mitades ordenadas de manera recursiva, lo que reduce el tiempo de ejecución a medida que el tamaño del arreglo disminuye.

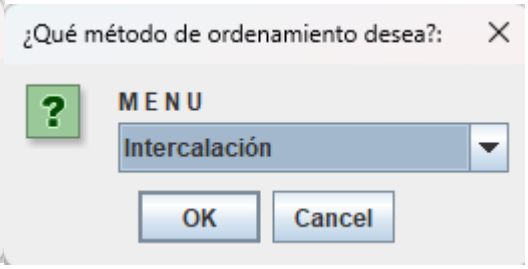
3. Peor caso: El peor caso ocurre cuando el arreglo está ordenado en orden inverso. En este escenario, el merge sort aún mantiene una complejidad temporal de O(n log n). Aunque puede parecer ineficiente ordenar un arreglo en orden inverso, el merge sort sigue siendo eficaz debido a su enfoque de dividir y fusionar en lugar de comparar y mover elementos individualmente. El peor caso se presenta cuando el algoritmo realiza todas las divisiones y fusiones necesarias para ordenar el arreglo.

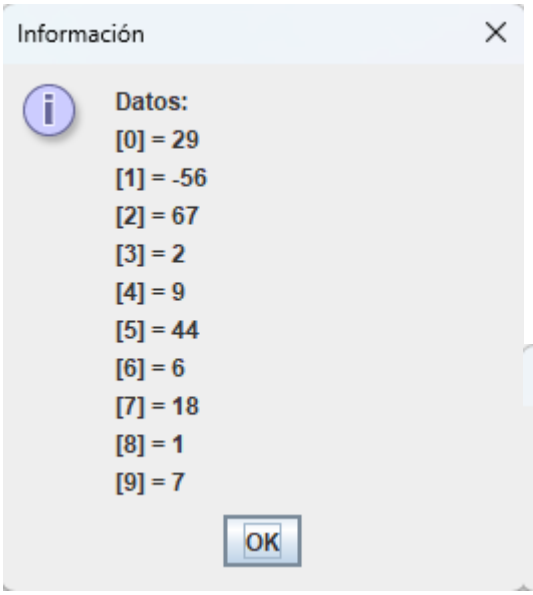
✓ Prueba de escritorio:

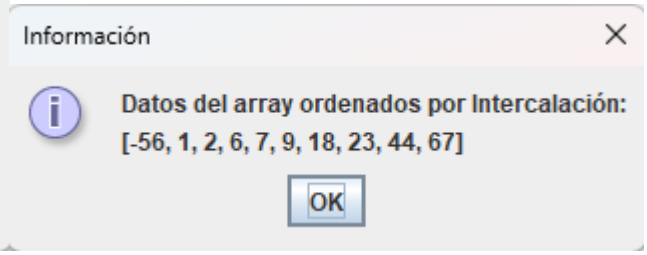












Mezcla directa.

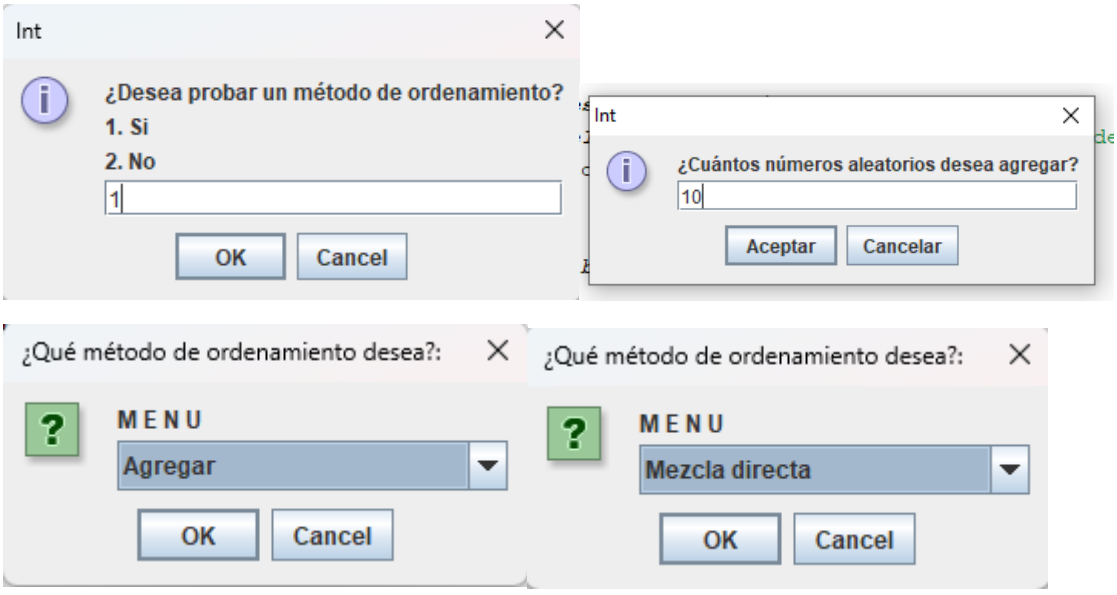
✓ Análisis de eficiencia:

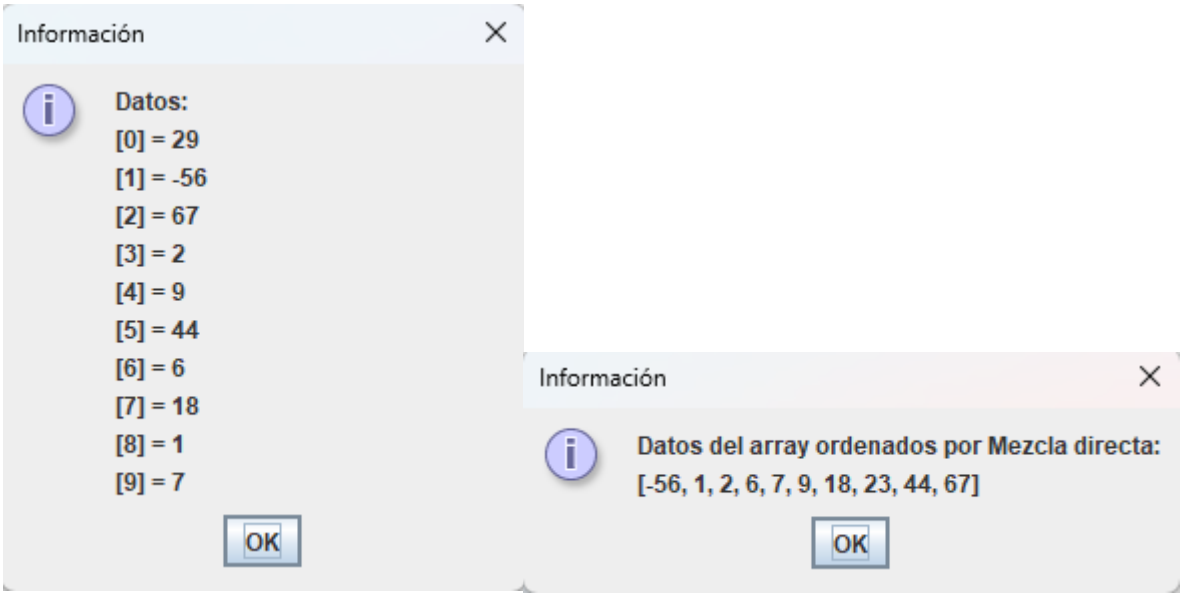
el método de ordenamiento por mezcla directa (merge sort directo) es eficiente en términos de tiempo en el caso promedio, con una complejidad temporal de O(n log n). Aunque tiene un peor caso con complejidad temporal de O(n^2), sigue siendo más eficiente que métodos de ordenamiento cuadráticos. Además, es un algoritmo estable y utiliza una cantidad razonable de memoria adicional. Estas características hacen que el merge sort directo sea ampliamente utilizado en la práctica para ordenar arreglos de diferentes tamaños.

✓ Análisis de los casos:

1. Mejor caso: El mejor caso ocurre cuando el arreglo ya está completamente ordenado. En este caso, el algoritmo realiza las divisiones y fusiones del arreglo sin necesidad de hacer intercambios entre elementos. Como resultado, el tiempo de ejecución del merge sort directo en el mejor caso es de O(n log n), donde "n" es el tamaño del arreglo. Esto se debe a que cada división reduce el tamaño del arreglo a la mitad, y se realizan log n divisiones. La fusión de las partes ordenadas requiere comparaciones, pero no se necesitan intercambios.
2. Caso promedio: El caso promedio del merge sort directo también tiene una complejidad temporal de O(n log n). Esto significa que, en promedio, el tiempo de ejecución crece de forma logarítmica con el tamaño del arreglo. El algoritmo divide el arreglo en partes más pequeñas y las fusiona en orden, utilizando comparaciones directas entre los elementos. A medida que el tamaño del arreglo disminuye, el número de comparaciones necesarias se reduce, lo que resulta en una eficiencia superior a otros métodos de ordenamiento cuadráticos como el bubble sort o el insertion sort.
3. Peor caso: El peor caso ocurre cuando el arreglo está ordenado en orden inverso. En esta situación, el merge sort directo requiere realizar el máximo número de comparaciones y movimientos de elementos. Cada comparación y movimiento se realiza en cada etapa de fusión. El peor caso tiene una complejidad temporal de O(n log n), donde "n" es el tamaño del arreglo. Aunque el peor caso tiene el mismo orden de complejidad que el mejor caso, el tiempo de ejecución real puede ser más lento debido a los movimientos de elementos requeridos en cada fusión.

✓ Prueba de escritorio:





Conclusión

En conclusión, los métodos de ordenamiento son herramientas fundamentales en el campo de las estructuras de datos. Proporcionan la capacidad de organizar eficientemente grandes volúmenes de información, permitiendo un acceso más rápido y eficiente a los datos almacenados. Al elegir un método de ordenamiento adecuado, es esencial considerar el tamaño del conjunto de datos, el tiempo y los recursos disponibles, así como las restricciones específicas del problema.

Además, la elección de un algoritmo de ordenamiento también puede depender del tipo de datos que se está ordenando. Algunos algoritmos son más eficientes para datos numéricos, mientras que otros son más adecuados para datos de texto o estructuras más complejas.

La elección correcta del algoritmo de ordenamiento es crucial para garantizar un procesamiento eficiente de los datos y mejorar el rendimiento general de las aplicaciones y sistemas que hacen uso de las estructuras de datos.

Bibliografía

Callejas, A. (3 de Enero de 2014). *El rincon de programacion*. Obtenido de elrinconprograues.blogspot.com: https://elrinconprograues.blogspot.com/2014/01/ordenacion-externa-mezcla-directa.html

Fernández, G. (18 de Enero de 2019). *LatteAndCode*. Obtenido de latteandcode.medium.com: https://latteandcode.medium.com/algoritmos-de-ordenaci%C3%B3n-quicksort-en-javascript-f064db39e6ad

Jindal, H. (25 de Febrero de 2021). *DelftStack*. Obtenido de www.delftstack.com: https://www.delftstack.com/es/tutorial/algorithm/heap-sort/

Jindal, H. (30 de Enero de 2023). *DelftStack*. Obtenido de https://www.delftstack.com/es/tutorial/algorithm/binary-sort/: https://www.delftstack.com/es/tutorial/algorithm/binary-sort/

Navarro, A. (30 de Septiembre de 2016). *JuncoTIC*. Obtenido de juncotic.com: https://juncotic.com/ordenamiento-de-burbuja-algoritmos-de-ordenamiento/

Pestaña Jiménez, E. A. (10 de April de 2015). *issuu*. Obtenido de issuu.com: https://issuu.com/ernestoalonsopestanajimenez/docs/metodo_burbuja_optimizado.pptx

GitHub. (2021). QuickSort Recursive Implementation in Java. Recuperado de https://gist.github.com/leonlipe/d64122e371e94a21ccb6eae2f01849c5

Procomsys. (2018, Junio 25). Caso práctico en Java: Ordenamiento de Burbuja (Bubble Sort). Recuperado de https://procomsys.wordpress.com/2018/06/25/caso-practico-en-java-ordenamiento-de-burbuja-bubble-sort/

Tutospoo. (s.f.). Ordenación rápida (Quicksort). Recuperado deTutospoo. (s.f.). Ordenación rápida (Quicksort). Recuperado de https://tutospoo.jimdofree.com/tutoriales-java/métodos-de-ordenación/ordenación-rápida-quicksort/