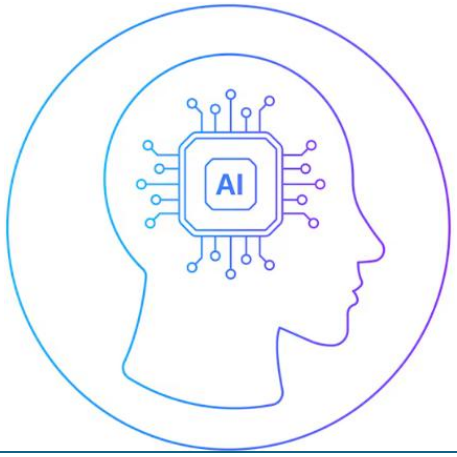

INTELIGENCIA ARTIFICIAL

Antonio de Jesús Covarrubias Sánchez.
Registro: 22110347
T-6E1
PRACTICA 2#

CENTRO DE ENSEÑANZA TECNICA INDUSTRIAL

MÉTODO DE ORDENAMIENTO QUICKSORT

Qué es QuickSort, qué hace y cómo lo hace

QuickSort es un algoritmo de ordenamiento muy popular y eficiente, que pertenece a la categoría de algoritmos de divide y vencerás. Fue desarrollado por el científico británico Tony Hoare en 1960. Desde su creación, ha sido una de las soluciones más utilizadas para ordenar listas de datos en muchos lenguajes de programación debido a su eficiencia en la práctica, especialmente cuando se compara con otros algoritmos como Burbuja o Selección.

QuickSort, como su nombre indica, es un algoritmo de ordenamiento rápido. El objetivo de este algoritmo es ordenar una lista de elementos de manera eficiente. A pesar de que existen muchos algoritmos de ordenamiento, QuickSort es conocido por su excelente rendimiento en la mayoría de los casos, superando a otros algoritmos tradicionales, como el ordenamiento por inserción o por burbuja, que tienen una complejidad computacional de $O(n^2)$.

La idea central de QuickSort es tomar un elemento de la lista como *pivote* y luego reordenar la lista de tal forma que los elementos menores que el pivote se ubiquen antes que él, y los elementos mayores se ubiquen después. Este proceso de dividir y conquistar se realiza recursivamente en las sublistas de elementos menores y mayores al pivote, hasta que toda la lista está ordenada.

Descripción del algoritmo

División y Conquista

El algoritmo QuickSort sigue el paradigma de "divide y vencerás". Este método se puede describir en los siguientes pasos:

1. **Seleccionar un pivote**: El primer paso es elegir un elemento de la lista para ser el pivote. Existen diferentes estrategias para elegir este pivote. Se puede seleccionar el primer elemento, el último elemento, el medio de la lista o incluso elegir un pivote aleatorio. La elección del pivote puede influir en el rendimiento del algoritmo.
2. **Reorganizar la lista**: Después de seleccionar el pivote, el siguiente paso es reorganizar los elementos en la lista de tal manera que todos los elementos menores que el pivote se ubiquen antes que él y los elementos mayores que el pivote se ubiquen después. Esto es conocido como la partición de la lista.
3. **Aplicar recursión**: El paso final es aplicar QuickSort recursivamente en las sublistas formadas por los elementos menores y mayores al pivote. Este proceso se repite hasta que las sublistas tengan uno o ningún elemento, lo que implica que están ordenadas.

Partición

La operación de partición es crucial en el algoritmo QuickSort. Existen diferentes métodos para llevar a cabo la partición, pero el más común es el siguiente:

- Se elige un pivote de la lista.
- Se recorren los elementos de la lista de izquierda a derecha, comparando cada elemento con el pivote.
- Si el elemento es menor que el pivote, se mueve a la parte izquierda de la lista.
- Si el elemento es mayor que el pivote, se mueve a la parte derecha.
- Una vez que todos los elementos han sido procesados, el pivote se coloca en su posición final, es decir, en el lugar donde todos los elementos a su izquierda son menores y todos los elementos a su derecha son mayores.

Este proceso de partición garantiza que, después de una ejecución, el pivote esté en su posición correcta y la lista esté dividida en dos partes: una con los elementos menores que el pivote y otra con los elementos mayores.

Recursión

Después de la partición, QuickSort se aplica recursivamente a las sublistas de elementos menores y mayores al pivote. Este paso se repite hasta que las sublistas sean lo suficientemente pequeñas para ser ordenadas de forma trivial (cuando tienen un solo elemento o ninguno).

Estrategias de selección del pivote

La eficiencia de QuickSort depende de cómo se elija el pivote. Si el pivote se selecciona de forma adecuada, el algoritmo tiene un rendimiento muy eficiente. Sin embargo, si el pivote se selecciona de manera ineficaz, el rendimiento puede empeorar considerablemente. Las principales estrategias para seleccionar un pivote son:

- **Pivote aleatorio**: Elegir un pivote aleatorio es una de las estrategias más comunes. Esto ayuda a evitar los peores casos que ocurren cuando la lista está ordenada o casi ordenada. La selección aleatoria mejora el rendimiento en la mayoría de los casos.
- **Pivote medio**: Seleccionar el pivote como el elemento medio de la lista es otra opción común. Aunque no garantiza el mejor rendimiento en todos los casos, es una estrategia razonable que da buenos resultados.
- **Pivote determinista**: En algunos casos, como listas que ya están ordenadas, puede ser más eficiente elegir el primer o el último elemento

como pivote. Sin embargo, esto puede llevar a un rendimiento muy deficiente en casos adversos.

Análisis de complejidad

El rendimiento de QuickSort depende de la calidad de la partición y de cómo se elige el pivote. Aquí se describen los principales casos de complejidad:

Mejor caso (O(n log n))

En el mejor de los casos, el pivote divide la lista de manera equilibrada en cada paso. Esto significa que la lista se divide en dos mitades, lo que da lugar a una complejidad de O(n log n), donde n es el número de elementos en la lista.

Peor caso ($O(n^2)$)

El peor caso ocurre cuando el pivote seleccionado es el elemento más pequeño o más grande en cada partición, lo que resulta en una partición desequilibrada. Esto puede ocurrir cuando la lista está ordenada o casi ordenada. En este caso, la complejidad es $O(n^2)$, ya que se realizan n particiones, y cada partición requiere O(n) tiempo.

Caso promedio (O(n log n))

En la práctica, el caso promedio es el más común y también es el caso más estudiado. Si se elige un pivote aleatorio o un pivote adecuado, el algoritmo tiende a tener una complejidad de O(n log n), que es eficiente incluso para listas grandes.

Ventajas y desventajas de QuickSort

Ventajas

- **Eficiencia**: QuickSort es uno de los algoritmos más rápidos en la mayoría de los casos. Su complejidad promedio es O(n log n), lo que lo hace eficiente incluso para listas grandes.
- **Enlace interno**: QuickSort no requiere memoria adicional para almacenar los datos, ya que es un algoritmo de ordenamiento en *in-place*, lo que significa que ordena la lista sin utilizar memoria adicional significativa.
- **Recursión**: La recursión hace que el algoritmo sea sencillo de implementar y comprender.

Desventajas

- **Peor caso**: En el peor de los casos, QuickSort puede ser ineficiente, especialmente si el pivote se selecciona de manera incorrecta. Esto puede ocurrir si la lista está ya ordenada o casi ordenada.

- **Uso de pila**: Dado que QuickSort utiliza recursión, puede consumir mucha memoria de pila si la lista es muy grande. En el peor de los casos, puede llegar a un nivel de recursión que cause un desbordamiento de pila.

Optimización de QuickSort

Existen varias técnicas para optimizar QuickSort y evitar su peor rendimiento:

- **Selección de pivote mejorada**: Seleccionar un pivote de manera más inteligente, como el uso de *median of three* (mediana de tres), puede mejorar el rendimiento al evitar particiones desbalanceadas.
- **Uso de un algoritmo de ordenamiento diferente para listas pequeñas**: En lugar de continuar con Quicksort en listas pequeñas (por ejemplo, con menos de 10 elementos), es más eficiente utilizar otro algoritmo, como *InsertionSort*, que tiene un rendimiento muy bueno en listas pequeñas.

MÉTODO DE ORDENAMIENTO DE BURBUJA (BUBBLESORT)

El algoritmo de ordenamiento de burbuja, conocido comúnmente como *Bubble Sort*, es uno de los algoritmos más simples y fáciles de entender en el campo de la informática y la programación. A pesar de su simplicidad, tiene una eficiencia relativamente baja en comparación con otros algoritmos de ordenación más avanzados, y generalmente se utiliza con fines educativos para ilustrar cómo funcionan los algoritmos de ordenación básicos. A continuación, se explicará en detalle qué es el método de ordenamiento de burbuja, cómo funciona, sus características, complejidades, ventajas y desventajas, y cuándo es apropiado usarlo.

El algoritmo de ordenamiento de burbuja es un algoritmo de comparación y intercambio que se utiliza para ordenar un conjunto de elementos. El principio básico detrás de este algoritmo es comparar los elementos adyacentes de la lista y, si están en el orden incorrecto, intercambiarlos. Este proceso se repite una y otra vez hasta que toda la lista está ordenada.

El término "burbuja" proviene de la forma en que los elementos más grandes "suben" hacia el final de la lista, como las burbujas en un líquido. Esto se debe a que, después de cada iteración, el mayor de los elementos no ordenados se mueve a su posición correcta, de manera similar a cómo una burbuja asciende hacia la superficie.

Funcionamiento de Bubble Sort

Proceso de Ordenación

Para comprender el algoritmo, consideremos una lista de números desordenados. El algoritmo comienza en el primer elemento de la lista y compara este elemento con el siguiente. Si el primer elemento es mayor que el siguiente, se intercambian. Este proceso continúa comparando y, si es necesario, intercambiando los elementos adyacentes hasta que el último elemento de la lista sea alcanzado. En ese momento, el mayor de los elementos se habrá "burbujeado" hasta la última posición de la lista.

Una vez que se ha completado una pasada por la lista, el algoritmo repite el proceso, pero esta vez no es necesario revisar el último elemento, ya que ya se encuentra en su posición correcta. Este procedimiento se repite, reduciendo gradualmente la cantidad de elementos que deben ser considerados en cada iteración, hasta que toda la lista esté ordenada.

Ejemplo

Supongamos que tenemos la siguiente lista de números desordenados:

[5, 2, 9, 1, 5, 6]

La primera pasada del algoritmo consistirá en comparar y, si es necesario, intercambiar los elementos adyacentes:

1. Compara 5 con 2, como 5 > 2, los intercambia, obteniendo [2, 5, 9, 1, 5, 6].
2. Compara 5 con 9, como 5 < 9, no se realiza ningún intercambio.
3. Compara 9 con 1, como 9 > 1, los intercambia, obteniendo [2, 5, 1, 9, 5, 6].
4. Compara 9 con 5, como 9 > 5, los intercambia, obteniendo [2, 5, 1, 5, 9, 6].
5. Compara 9 con 6, como 9 > 6, los intercambia, obteniendo [2, 5, 1, 5, 6, 9].

Después de esta primera pasada, el número más grande, 9, ha sido colocado en su posición correcta. Ahora se repetirá el proceso, pero con una lista más pequeña:

[2, 5, 1, 5, 6]

El algoritmo continuará así, realizando pasadas sucesivas por la lista hasta que no haya más intercambios necesarios.

Complejidad Temporal de Bubble Sort

Mejor Caso

El mejor caso de Bubble Sort ocurre cuando la lista ya está ordenada. En este caso, el algoritmo solo realiza una pasada a través de la lista y no realiza ningún intercambio. Esto lleva a una complejidad temporal de O(n), donde n es el número de elementos en la lista. Este caso se puede optimizar si se agrega una verificación que indique si se realizaron intercambios en la pasada; si no se realizó ningún intercambio, el algoritmo puede detenerse antes de completar todas las pasadas.

Peor Caso

El peor caso de Bubble Sort ocurre cuando la lista está completamente desordenada, lo que lleva al algoritmo a realizar el máximo número de intercambios. En este caso, el algoritmo realizará n-1 pasadas, cada una con un número decreciente de comparaciones. La complejidad temporal en el peor caso es $O(n^2)$, donde n es el número de elementos en la lista. Esto se debe a que el algoritmo realiza n-1 comparaciones en la primera pasada, n-2 en la segunda, y así sucesivamente.

Caso Promedio

El caso promedio también tiene una complejidad de $O(n^2)$, ya que el algoritmo realiza comparaciones e intercambios entre los elementos de la lista. Sin embargo, en la práctica, puede haber menos intercambios que en el peor caso, pero la cantidad de comparaciones sigue siendo cuadrática en promedio.

Ventajas y Desventajas del Método de Ordenamiento de Burbuja

Ventajas

- **Simplicidad**: El algoritmo de ordenamiento de burbuja es extremadamente simple de entender e implementar, lo que lo convierte en un buen punto de partida para aprender sobre algoritmos de ordenación.
- **In-place**: Bubble Sort no requiere memoria adicional para almacenar una copia de la lista, ya que realiza los intercambios directamente sobre la lista original. Esto lo convierte en un algoritmo "in-place".
- **Mejorado con optimización**: Si se agrega una verificación para detectar si la lista ya está ordenada, Bubble Sort puede ser ligeramente más eficiente en algunos casos, especialmente en listas que ya están parcialmente ordenadas.

Desventajas

- **Ineficiencia**: El principal inconveniente de Bubble Sort es su ineficiencia en listas grandes. Con una complejidad temporal de $O(n^2)$, se vuelve rápidamente ineficaz en comparación con otros algoritmos de ordenación más rápidos, como el QuickSort o el MergeSort.
- **No es adecuado para listas grandes**: Debido a su complejidad cuadrática, Bubble Sort no es adecuado para ordenar listas grandes. Otros algoritmos de ordenación como QuickSort, MergeSort y HeapSort ofrecen un rendimiento significativamente mejor.
- **Rendimiento en el peor caso**: El algoritmo realiza muchas comparaciones e intercambios innecesarios, incluso cuando la lista está casi ordenada.

Comparación con Otros Algoritmos de Ordenación

A pesar de ser fácil de entender, el ordenamiento de burbuja es superado en eficiencia por muchos otros algoritmos de ordenación. Aquí se presenta una comparación de Bubble Sort con algunos otros algoritmos populares:

- **QuickSort**: QuickSort tiene una complejidad temporal promedio de $O(n \log n)$, lo que lo convierte en un algoritmo mucho más eficiente que Bubble Sort en la mayoría de los casos.
- **MergeSort**: MergeSort también tiene una complejidad temporal de $O(n \log n)$ y es un algoritmo eficiente y estable, que a menudo se utiliza en aplicaciones que requieren una alta fiabilidad en la ordenación de grandes conjuntos de datos.
- **Insertion Sort**: Aunque Insertion Sort también tiene una complejidad de $O(n^2)$ en el peor caso, en algunos casos, como cuando los datos ya están casi ordenados, puede ser más eficiente que Bubble Sort.

Dado que el algoritmo de ordenamiento de burbuja es muy ineficiente para listas grandes, rara vez se utiliza en aplicaciones del mundo real para ordenar grandes volúmenes de datos. Sin embargo, es útil en los siguientes casos:

- **Listas pequeñas**: En situaciones donde se requiere ordenar un número reducido de elementos, Bubble Sort puede ser suficiente y sencillo de implementar.
- **Enseñanza**: Debido a su simplicidad, el algoritmo es ideal para enseñar a los estudiantes sobre los conceptos básicos de los algoritmos de ordenación y la complejidad computacional.

MÉTODO DE ORDENAMIENTO POR INSERCIÓN

El Método de Ordenamiento por Inserción (conocido en inglés como Insertion Sort) es un algoritmo de ordenación que organiza los elementos de una lista o array en un orden específico (generalmente ascendente o descendente). Es uno de los algoritmos más sencillos y fáciles de entender en el campo de la informática y la programación, aunque no es el más eficiente en términos de tiempo para listas grandes. En este artículo, analizaremos qué es el ordenamiento por inserción, cómo funciona, sus ventajas y desventajas, y su implementación.

¿Qué es?

El ordenamiento por inserción es un algoritmo que ordena una lista comparando elementos adyacentes y colocándolos en su posición correcta mediante inserciones sucesivas. Imagina cómo ordenarías un conjunto de cartas en tu mano: al principio, solo tienes una carta, que está ordenada. Luego, tomas una carta más y la insertas en el lugar adecuado en la secuencia ya ordenada. Repites este proceso con todas las cartas hasta que toda la mano esté ordenada.

En términos de algoritmo, esto significa que el ordenamiento por inserción comienza con el segundo elemento de la lista y lo inserta en su lugar correcto en la parte izquierda de la lista ordenada. Luego, pasa al siguiente elemento y lo coloca en el lugar adecuado en la parte ordenada de la lista, repitiendo este proceso hasta que toda la lista esté ordenada.

¿Cómo Funciona?

El algoritmo de ordenamiento por inserción funciona mediante una serie de pasos:

1. **Comienzo con el segundo elemento**: La idea central del algoritmo es que la primera parte de la lista, en todo momento, está ordenada. Al principio, solo hay un elemento ordenado, es decir, el primer elemento de la lista.
2. **Inserción del siguiente elemento en la parte ordenada**: Luego, el algoritmo toma el segundo elemento de la lista y lo compara con el primer elemento. Si es menor que el primer elemento, lo inserta en la posición correcta (que sería al principio de la lista). Si no, lo deja en su lugar.
3. **Avance a la siguiente posición**: El algoritmo avanza al siguiente elemento de la lista y lo inserta en la posición correcta de la parte ordenada de la lista.
4. **Repetición del proceso**: Este proceso se repite para cada elemento de la lista, moviendo gradualmente el elemento a su lugar adecuado en la parte ordenada.

Ejemplo

Consideremos el siguiente ejemplo para entender cómo funciona el algoritmo de inserción paso a paso:

Lista original: [6, 2, 8, 4, 3]

1. **Primer paso**: Comenzamos con el primer elemento (6), que ya está ordenado. Luego tomamos el segundo elemento (2).
    - Comparamos 2 con 6. Como 2 < 6, movemos 6 a la derecha, y colocamos 2 en la primera posición.
    - La lista ahora es: [2, 6, 8, 4, 3]
2. **Segundo paso**: Tomamos el tercer elemento (8).
    - Comparamos 8 con 6. Como 8 > 6, no hacemos nada y avanzamos al siguiente paso.
    - La lista sigue siendo: [2, 6, 8, 4, 3]
3. **Tercer paso**: Tomamos el cuarto elemento (4).
    - Comparamos 4 con 8. Como 4 < 8, movemos 8 a la derecha.
    - Comparamos 4 con 6. Como 4 < 6, movemos 6 a la derecha.
    - Colocamos 4 en la posición correspondiente.
    - La lista ahora es: [2, 4, 6, 8, 3]
4. **Cuarto paso**: Tomamos el quinto elemento (3).
    - Comparamos 3 con 8. Como 3 < 8, movemos 8 a la derecha.
    - Comparamos 3 con 6. Como 3 < 6, movemos 6 a la derecha.
    - Comparamos 3 con 4. Como 3 < 4, movemos 4 a la derecha.
    - Colocamos 3 en la posición correspondiente.
    - La lista ahora es: [2, 3, 4, 6, 8]

Al final, la lista está completamente ordenada.

Análisis de Complejidad

El ordenamiento por inserción tiene una complejidad temporal de $O(n^2)$ en el peor de los casos, donde **n** es el número de elementos en la lista. Esto ocurre cuando la lista está ordenada de forma descendente, lo que obliga al algoritmo a realizar una cantidad máxima de comparaciones e intercambios para cada elemento.

Sin embargo, si la lista ya está parcialmente ordenada o completamente ordenada, el tiempo de ejecución puede ser mucho más rápido, alcanzando **O(n)**, donde solo se requieren comparaciones sin intercambios.

Complejidad Temporal

- **Mejor caso (O(n))**: Si la lista ya está ordenada o está casi ordenada, solo se necesitarán comparaciones, y el algoritmo no realizará intercambios.
- **Peor caso ($O(n^2)$)**: Si la lista está completamente desordenada, el algoritmo realizará una gran cantidad de comparaciones e intercambios.
- **Caso promedio ($O(n^2)$)**: En la mayoría de los casos, el algoritmo realizará una cantidad moderada de comparaciones e intercambios.

Complejidad Espacial

La complejidad espacial del ordenamiento por inserción es $O(1)$, ya que no necesita espacio adicional significativo para ordenar la lista. Todo el trabajo se realiza en el lugar, sin la necesidad de estructuras de datos adicionales, lo que lo convierte en un algoritmo in-place.

Ventajas

1. **Simplicidad**: El algoritmo es muy fácil de entender e implementar, lo que lo convierte en una opción popular para pequeñas listas o como parte de algoritmos más complejos.
2. **Eficiente para listas pequeñas o casi ordenadas**: En listas pequeñas o listas que ya están parcialmente ordenadas, el algoritmo es bastante eficiente, ya que su complejidad temporal en el mejor caso es **O(n)**.
3. **Estabilidad**: El algoritmo es estable, lo que significa que mantiene el orden relativo de los elementos con el mismo valor.
4. **Eficiencia en la memoria**: La complejidad espacial del algoritmo es **O(1)**, lo que significa que no necesita memoria adicional significativa.

Desventajas

Ineficiencia para listas grandes: En listas grandes, el algoritmo se vuelve ineficiente debido a su complejidad temporal de **O(n²)**. Otros algoritmos de ordenación como **Merge Sort** o **Quick Sort** son más eficientes en estos casos.

1. **No apto para datos no ordenados en gran medida**: Si la lista está muy desordenada, el algoritmo se vuelve mucho más lento en comparación con otros métodos más eficientes.
2. **Requiere muchas comparaciones e intercambios**: En el peor de los casos, el algoritmo realiza muchas comparaciones e intercambios, lo que lo hace menos eficiente en situaciones específicas.

Comparación con Otros Algoritmos de Ordenación

En comparación con otros algoritmos de ordenación como Quick Sort, Merge Sort y Bubble Sort, el insertion sort es generalmente menos eficiente en términos de tiempo para listas grandes. Sin embargo, para listas pequeñas o casi ordenadas, puede ser una opción excelente debido a su simplicidad y eficiencia en memoria.

- **Quick Sort** tiene una complejidad temporal promedio de **O(n log n)**, mucho mejor que **O(n²)** del ordenamiento por inserción.
- **Merge Sort** también tiene una complejidad de **O(n log n)** en el peor de los casos, y es más eficiente que el ordenamiento por inserción para listas grandes.
- **Bubble Sort**, aunque también tiene una complejidad de **O(n²)**, es menos eficiente que el ordenamiento por inserción debido a la cantidad de intercambios innecesarios.

MÉTODO DE ORDENAMIENTO POR SELECCIÓN

El método de ordenamiento por selección o Selection Sort es un algoritmo clásico de ordenamiento en el ámbito de la informática. Es conocido por su sencillez y por su enfoque basado en la idea de seleccionar el elemento más pequeño (o más grande, dependiendo del orden de la ordenación) y colocarlo en la posición correcta dentro de la lista. A pesar de ser simple, no es eficiente para listas grandes debido a su alto costo computacional. Sin embargo, es fundamental para el aprendizaje de algoritmos de ordenamiento y se usa a menudo en contextos educativos para ilustrar conceptos básicos de algoritmos.

El método de ordenamiento por selección es un algoritmo de comparación que funciona seleccionando de manera repetida el elemento más pequeño (o el más grande) de la lista desordenada y colocándolo en su posición correcta, hasta que toda la lista está ordenada.

¿Cómo Funciona?

Para explicar cómo funciona el algoritmo, primero necesitamos entender cómo se realiza cada iteración:

1. **Inicialización**: El algoritmo comienza con la lista desordenada y selecciona un índice que servirá como punto de referencia para encontrar el siguiente valor a ordenar.
2. **Selección del mínimo**: En cada paso del algoritmo, selecciona el elemento más pequeño de la lista que aún no ha sido ordenado. Esto se hace iterando sobre la parte no ordenada de la lista.
3. **Intercambio**: Luego de encontrar el elemento mínimo, se intercambia con el elemento en la posición actual del índice de la iteración. Esto asegura que el mínimo se coloca en su posición correcta.
4. **Repetición**: El algoritmo continúa este proceso para el resto de la lista. Cada iteración reduce el tamaño de la parte no ordenada de la lista en uno.
5. **Terminación**: Cuando el algoritmo llega al final de la lista, la lista estará ordenada.

Ejemplo

Consideremos el siguiente ejemplo con la lista desordenada:

[64, 25, 12, 22, 11]

El proceso de ordenamiento por selección se lleva a cabo de la siguiente manera:

1. **Primera iteración**:
    - Seleccionamos el elemento más pequeño en la lista completa (el número 11).
    - Intercambiamos 64 con 11.
    - La lista ahora se ve así: [11, 25, 12, 22, 64].
2. **Segunda iteración**:
    - Ahora, consideramos la lista desde el segundo elemento hasta el último (es decir, [25, 12, 22, 64]).
    - El mínimo de esta sublista es 12.
    - Intercambiamos 25 con 12.
    - La lista ahora se ve así: [11, 12, 25, 22, 64].
3. **Tercera iteración**:
    - Consideramos la sublista [25, 22, 64].
    - El mínimo de esta sublista es 22.
    - Intercambiamos 25 con 22.
    - La lista ahora se ve así: [11, 12, 22, 25, 64].
4. **Cuarta iteración**:
    - Finalmente, consideramos la sublista [25, 64].
    - El mínimo es 25, por lo que no hay necesidad de intercambiar nada.
    - La lista ya está ordenada: [11, 12, 22, 25, 64].

Características del Algoritmo de Selección

1. **Estabilidad**: El algoritmo de selección **no es estable**. Esto significa que si hay dos elementos con el mismo valor, no se garantiza que mantendrán su orden relativo original en la lista ordenada.
2. **Complejidad temporal**: La complejidad temporal del algoritmo es **$O(n^2)$** en el peor, mejor y promedio de los casos. Esto se debe a que para cada elemento de la lista, el algoritmo recorre todos los elementos restantes en busca del mínimo.
3. **Espacio de almacenamiento**: El espacio de almacenamiento es **O(1)**, lo que significa que el algoritmo es **in-place** (no requiere espacio adicional significativo más allá de la lista original).
4. **Uso de intercambios**: El algoritmo realiza un número de intercambios, pero en comparación con otros algoritmos como el bubble sort, el número de intercambios es menor. Sin embargo, sigue siendo considerablemente alto en listas grandes.
5. **Mejor caso**: En su mejor caso, el algoritmo de selección realiza el mismo número de comparaciones que en el peor caso, ya que siempre debe recorrer el resto de la lista para encontrar el mínimo.

Propiedades y Consideraciones del Algoritmo de Selección

Complejidad del Algoritmo

La principal desventaja del algoritmo de selección es su complejidad temporal. Como se mencionó anteriormente, es $O(n^2)$. Esto lo convierte en un algoritmo ineficiente para listas grandes, especialmente si se compara con algoritmos más avanzados como QuickSort o MergeSort, que tienen una complejidad promedio de $O(n \log n)$.

La razón de esta complejidad es que el algoritmo realiza una búsqueda lineal en la lista para encontrar el elemento mínimo en cada iteración, lo que toma n-1 comparaciones en la primera iteración, n-2 en la siguiente, y así sucesivamente hasta llegar a 1 comparación en la última iteración. El número total de comparaciones es la suma de estos números, lo que da como resultado una complejidad cuadrática.

Ventajas y Desventajas

Ventajas:

1. **Simplicidad**: El algoritmo es fácil de entender y de implementar.
2. **Uso de memoria constante**: A diferencia de otros algoritmos de ordenamiento como MergeSort, el algoritmo de selección tiene una complejidad espacial O(1), lo que significa que no requiere memoria adicional significativa.

Desventajas:

1. **Ineficiencia**: En listas grandes, el algoritmo puede ser muy lento debido a su complejidad cuadrática **$O(n^2)$**.
2. **No estable**: No mantiene el orden de los elementos iguales, lo que puede ser una desventaja en algunos contextos.

Comparación con Otros Algoritmos de Ordenamiento

El algoritmo de selección, debido a su simplicidad, es a menudo utilizado en contextos donde el rendimiento no es una preocupación principal o cuando las listas para ordenar son pequeñas. Sin embargo, existen algoritmos más eficientes para el ordenamiento, como:

- **MergeSort**: Con una complejidad de **$O(n \log n)$**, MergeSort es mucho más rápido para listas grandes que el algoritmo de selección. Además, es un algoritmo estable.
- **QuickSort**: Otro algoritmo con **$O(n \log n)$** en promedio. Aunque no es estable, QuickSort es muy eficiente en la práctica debido a sus características de partición.
- **Insertion Sort**: Aunque también tiene una complejidad cuadrática, Insertion Sort es más eficiente que Selection Sort en listas pequeñas o casi ordenadas.