Instituto Politécnico Nacional
Escuela Superior de Cómputo

Práctica 01: Algoritmos de ordenamiento

Materia: Algoritmos y Estructuras de Datos
Profesor: Martínez Edgardo Franco

Alumno: González Joshua

Grupo: 2CM8
Fecha: 31/03/2023 ISC

Índice

Introducción

¿Qué es el ordenamiento? ¿Por qué es necesario en la computación? ¿Cuáles tipos de algoritmos existen? ¿Cuál es el algoritmo más eficiente e ineficiente? Son las preguntas fundamentales para partir con este reporte de práctica y comprender lo que es el ordenamiento en la computación.

La ordenación de datos es una tarea fundamental que se realiza en una amplia variedad de aplicaciones. Los algoritmos de ordenamiento son programas diseñados para ordenar un conjunto de datos en un orden específico, ya sea de manera ascendente o descendente. La implementación de estos algoritmos es una práctica común en la programación, especialmente en la programación de sistemas y aplicaciones donde el rendimiento y la eficiencia son esenciales, pues siempre será más fácil trabajar con una colección ordenada.

Existen muchos algoritmos de ordenamiento diferentes, cada uno con sus propias ventajas y desventajas en términos de tiempo de ejecución, espacio auxiliar o uso de memoria y complejidad del algoritmo. Algunos de los algoritmos de ordenamiento más populares incluyen el ordenamiento de burbuja, el ordenamiento por selección, el ordenamiento por inserción, el ordenamiento rápido y el ordenamiento por mezcla, y son los que veremos en esta práctica, además de una “variante” mejorada del algoritmo por selección, el algoritmo de ordenamiento Shell.

La implementación de algoritmos de ordenamiento puede ser una tarea desafiante, ya que requiere un conocimiento profundo de los algoritmos y su funcionamiento interno. Sin embargo, la implementación adecuada de algoritmos de ordenamiento puede tener un gran impacto en el rendimiento y la eficiencia de un programa, lo que lo convierte en una habilidad valiosa para cualquier programador.

Realizamos nuestra “Práctica 01: algoritmos de ordenamiento”, la cual tiene como objetivo la comprensión, análisis y evaluación de los diferentes algoritmos de ordenamiento para la solución de problemas, identificando la importancia de comprender los distintos algoritmos y su funcionamiento, además de que realizaremos un análisis de cuáles son los algoritmos más ineficientes en cuanto a memoria y tiempo de ejecución, comparando los datos con tablas y gráficas donde realizaremos pruebas con distintos tipos de datos.

Marco teórico

¿Qué es un algoritmo?

La palabra algoritmo toma su nombre de Al-khowarizimi, matemático y astrónomo del siglo IX quien, al escribir un tratado sobre manipulación de números y ecuaciones, el Kitab al-jabr w’almugabala, usó en gran medida la noción de lo que se conoce hoy como algoritmo.

Un algoritmo es una secuencia finita “bien definida” de tareas “bien definidas”, cada una de las cuales se puede realizar con una cantidad de recursos finitos. Aunque los recursos que debe utilizar cada tarea deben ser finitos estos no están limitados, es decir, si una tarea bien definida requiere una cantidad inmensa (pero finita) de algún recurso para su realización, dicha tarea puede formar parte de un algoritmo. Además, se dice que una

secuencia de tareas esta 'bien definida' si se sabe el orden exacto de ejecución de cada una de las mismas.

Un algoritmo es una lista precisa de pasos, su orden de ejecución será casi siempre crítico para su funcionamiento. Se asume que las instrucciones se enumeran explícitamente.

Ejemplos de un algoritmo:

- Cuando vas a comer.
  1. Compras o haces de comer.
  2. Te lavas las manos.
  3. Sirves la comida.
  4. Te sientas en la mesa.
  5. Comes.
- Cuando te vistes por la mañana.
  1. Te despiertas.
  2. Te lavas la cara.
  3. Escoges la ropa que te vas a poner.
  4. Escoges los zapatos.
  5. Te quitas el pijama.
  6. Te pones el pantalón.
  7. Te pones la playera.
  8. Te pones las calcetas.
  9. Luego los zapatos
  10. Listo.

Características de un algoritmo

Las características de un algoritmo existen según su tipo, diferentes características particulares que un algoritmo debe poseer, pero en la generalidad se deberán contemplar las siguientes características:

- Entrada: Definir lo que necesita el algoritmo.
- Salida: Define lo que produce.
- No ambiguo: explicito, siempre sabe que comando ejecutar.
- Finito: El algoritmo termina en un número finito de pasos.
- Correcto: La solución debe ser correcta (aceptable).
- Efectividad: Cada instrucción se completa en tiempo finito.
- General: Debe contemplar todos los casos de entrada.

"Un algoritmo es un procedimiento para resolver un problema, cuyos pasos son concretos y no ambiguos. El algoritmo debe ser correcto, de longitud finita y debe terminar para todas las entradas".

Los algoritmos en computación se clasifican de diversas maneras, según el camino a seguir, según su tipo de resultado final, según su campo de injerencia o el tipo de problemas que resuelve, según su complejidad, su esquema de diseño, el tipo de problema específico que ataca, etc.

El ordenamiento es el proceso de reordenar un conjunto de objetos en un orden especifico. El propósito de la ordenación es facilitar la búsqueda de elementos en el conjunto ordenado.

Existen muchos algoritmos de ordenamiento, siendo la diferencia entre ellos las ventajas de unos sobre otros en la eficiencia de tiempo de ejecución.

El problema de ordenamiento

El problema de ordenamiento puede establecerse mediante la siguiente acción: dados los elementos $a_1, a_2, \dots, a_n$. Ordenar consiste en permutar esos elementos en un orden $a_{k1}, a_{k2}, \dots, a_{kn}$ tal que dada una función de ordenamiento se verifique $f(a_{k1}) \leq f(a_{k2}), \dots \leq f(a_{kn})$. Normalmente la función de ordenamiento no es evaluada de acuerdo con una regla de computación determinada, pero se guarda como un componente explicito (campo) de cada elemento (item). Ese valor de ese campo se llama “la llave del elemento”.

Si un conjunto de datos a ordenar tiene duplicadas y se mantiene el orden relativo de los datos con clave repetida en el conjunto de datos original, el ordenamiento que se realice se dice que es estable.

Se entiende que los métodos de ordenamiento buscan un uso eficiente de la memoria por lo que las permutaciones de elementos en la mayoría de los algoritmos que sean capaces de ordenar los elementos se prefiere hacer in situ, es decir, usando el mismo contenido original.

El problema de ordenamiento se refiere a la tarea de ordenar una colección de elementos en un cierto orden específico. La mayoría de las veces, el objetivo es ordenar los elementos en orden ascendente o descendente en función de alguna clave o criterio de ordenamiento definido. El ordenamiento se utiliza comúnmente en una amplia variedad de aplicaciones, como la organización de datos en bases de datos, la presentación de información en orden, la búsqueda de elementos en una colección ordenada y la optimización de algoritmos de procesamiento de datos. Hay varios algoritmos de ordenamiento disponibles, cada uno con sus propias ventajas y desventajas en términos de complejidad de tiempo y espacio. La elección del algoritmo de ordenamiento adecuado depende del tamaño de la colección de datos, la complejidad de los datos y el contexto de la aplicación.

Ordenamiento en computación

Los ordenamientos de computación se pueden realizar a elementos en:

- Memoria principal: Ordenar elementos en la memoria en tiempo de ejecución (se o0btiene un acceso aleatorio a los elementos, pero se desea poder localizarlos más fácil y por ello se ordenan).
- Memoria secundaria: Ordenar archivos en disco o información que no cuenta con un acceso aleatorio en memoria.
- En computación ordenar, es necesario para la mayoría de los algoritmos que almacenan, editan o acceden a información, los elementos a ordenar representan información a procesar que en conjunto tienen una relación o una dependencia (campo clave), que permite crear una relación de orden entre estos.

Ordenamiento de burbuja

La idea principal de este algoritmo de ordenamiento es recorrer un conjunto de datos comparando pares de elementos, si el primero es mayor al segundo, invertir sus lugares. Se repite el procedimiento hasta recorrer todo el conjunto. [1]

Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesitan más intercambios, lo que significa que la lista está ordenada.

El método de burbuja es uno de los más simples, es tan simple como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor que otro, entonces los intercambia de posición. Este algoritmo obtiene su nombre de la forma con la que se suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También conocido como el método de intercambio directo.

Ventajas del ordenamiento burbuja:

- Es muy fácil y simple de implementar.
- No requiere ningún espacio de memoria adicional.
- Es un algoritmo de ordenación estable, lo que significa que los elementos con el mismo valor clave mantienen su orden relativo en la salida ordenada.

Desventajas del ordenamiento burbuja:

- Es el algoritmo más lento e ineficiente entre los algoritmos de ordenamiento.
- Bubble sort tiene una complejidad de tiempo de O (n ^ 2) que lo hace muy lento para grandes conjuntos de datos.
- No es eficiente para grandes conjuntos de datos, porque requiere varias pasadas a través de los datos.

Mejor caso para el ordenamiento burbuja

Su mejor caso es cuando los elementos ya se encuentran ordenados y no es necesario realizar ninguna operación o intercambio, se utiliza la variante de burbuja optimizada, para que una vez que se haga una pasada completa o se recorra el arreglo, y se verifique que éste ya se encuentra ordenado, se detenga el algoritmo, lo que resultaría en una complejidad de $O(n)$.

Análisis del peor caso para el ordenamiento burbuja

La complejidad del algoritmo de ordenamiento burbuja en el peor de los casos es $O(n^2)$, donde n es el número de elementos a ordenar. Esto ocurre cuando el conjunto de elementos está en orden inverso, lo que significa que cada elemento tiene que intercambiarse con cada uno de los otros elementos antes de que el conjunto esté completamente ordenado.

En este caso, el algoritmo de burbuja realiza n-1 pasadas completas por el conjunto de elementos, donde en cada pasada se realizan n-i comparaciones e intercambios, siendo i el número de la pasada actual. Por lo tanto, el número total de comparaciones e intercambios que se realizan en el peor caso es:

$$Suma\ de\ Gauss: \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Esta complejidad es cuadrática en el tamaño del conjunto de elementos, lo que significa que el tiempo de ejecución del algoritmo puede aumentar significativamente para conjuntos grandes de datos. Por esta razón, el algoritmo de burbuja no se utiliza comúnmente en aplicaciones prácticas que requieren un alto rendimiento en la ordenación de grandes conjuntos de datos.

Ordenamiento por Selección

Es un algoritmo de ordenación simple y eficiente que funciona seleccionando repetidamente el elemento más pequeño (o más grande) de la parte no ordenada de la lista y moviéndolo a la parte ordenada de la lista. El algoritmo selecciona repetidamente el elemento más pequeño (o más grande) de la parte no ordenada de la lista y lo intercambia con el primer elemento de la parte no ordenada. Este proceso se repite para la porción restante sin ordenar de la lista hasta que se ordena toda la lista.

Se basa en buscar el mínimo elemento de la lista e intercambiarlo con el primero, después busca el siguiente mínimo en el resto de la lista y lo intercambia con el segundo, así sucesivamente.

Ventajas del ordenamiento por selección:

- Simple y fácil de entender.
- Conserva el orden relativo de los elementos con claves iguales, lo que significa que es estable.
- Es adaptable a varios tipos de tipos de datos.
- No requiere ninguna memoria especial ni estructuras de datos auxiliares, lo que la convierte en una solución ligera.

Desventajas del ordenamiento por selección:

- La ordenación por selección tiene una complejidad de tiempo de O(n^2) en el peor de los casos y en el promedio.
- No funciona bien en grandes conjuntos de datos.
- Tiene muchas operaciones de escritura, lo que genera un rendimiento deficiente en sistemas con almacenamiento lento.

Análisis de complejidad en el peor de los casos para el ordenamiento por selección

Ocurre cuando el conjunto de elementos está en orden inverso, lo que significa que se deben realizar $n-1$ intercambios en la primera pasada completa por el conjunto de elementos, $n-2$ intercambios en la segunda pasada completa, y así sucesivamente hasta que se realiza un intercambio en la última pasada completa, lo que resulta en una complejidad del algoritmo de ordenamiento por selección en el peor de los casos de $O(n^2)$.

En el peor caso, el número total de intercambios que se realizan es:

$$Suma\ de\ Gauss: \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Que es cuadrático en el tamaño del conjunto de elementos. Por lo tanto, el tiempo de ejecución del algoritmo puede aumentar significativamente para conjuntos grandes de datos.

Análisis de complejidad en el mejor de los casos para el ordenamiento por selección

En el mejor caso, se realiza exactamente un intercambio en cada pasada completa por el conjunto de elementos, esto ocurre cuando el conjunto ya está ordenado, pero posiblemente no de forma creciente. Debido a que solo se realiza un intercambio por pasada en el mejor caso, el número total de intercambios es $n-1$, que es la cantidad mínima posible de intercambios para ordenar cualquier conjunto de elementos. Sin embargo, la complejidad $O(n^2$ se mantiene, ya que en el mejor caso todavía se realizan $n-1$ pasadas completas por el conjunto de elementos, y en cada pasada se deben realizar $n-i$ comparaciones, siendo i el número de la pasada actual.

Espacio auxiliar para el ordenamiento por selección

$O(1)$ ya que la única memoria adicional utilizada es para variables temporales mientras se intercambian dos valores en Array. La ordenación por selección nunca hace más que intercambios $O(N)$ y puede ser útil cuando la escritura en memoria es una operación costosa.

Ordenamiento por inserción

El ordenamiento por inserción es un algoritmo de ordenamiento simple que funciona comparando cada elemento de una lista o arreglo con los elementos que le preceden, y si se encuentra uno que sea menor (o mayor, dependiendo del orden deseado), se intercambia con él. Este proceso se repite hasta que la lista esté completamente ordenada. Este algoritmo es útil para ordenar pequeñas cantidades de datos o para agregar nuevos elementos a una lista que ya está ordenada, ya que su complejidad de tiempo es de O(n2) en el peor de los casos. A pesar de esto, es un algoritmo eficiente en términos de uso de memoria, ya que solo se necesitan variables auxiliares para el intercambio de elementos.

Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas enumeradas en forma arbitraria.

Inicialmente se tiene un solo elemento, que obviamente es conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor o cuando ya no se encuentran elementos. En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

Ventajas del ordenamiento por inserción

- Es un algoritmo simple y fácil de entender, por lo que es una buena opción para conjuntos pequeños de datos.

- Tiene una complejidad de tiempo de $O(n^2)$ en el peor caso, lo que lo hace eficiente para conjuntos de datos pequeños y medianos.
- Tiene un uso eficiente de la memoria, ya que no requiere espacio adicional para almacenar los datos a ordenar.
- Es estable, lo que significa que los elementos con valores iguales se mantendrán en su orden relativo después de la ordenación.

Desventajas del ordenamiento por inserción

- La complejidad de tiempo del algoritmo puede ser cuadrática en el peor caso, lo que significa que no es adecuado para conjuntos de datos grandes.
- No es eficiente para conjuntos de datos que ya están en orden inverso o parcialmente ordenados, ya que se requiere un número significativo de comparaciones e intercambios.
- El número de comparaciones y operaciones de intercambio puede aumentar significativamente para conjuntos grandes de datos, lo que puede ralentizar el tiempo de ejecución.
- Tiene una complejidad de tiempo de O(n) en el mejor caso, lo que significa que es menos eficiente que otros algoritmos de ordenamiento como el ordenamiento por mezcla o el ordenamiento rápido en casos ideales.

Análisis de complejidad en el peor de los casos para el ordenamiento por inserción

En el peor de los casos, el algoritmo de ordenamiento por inserción tiene una complejidad de tiempo de $O(n^2)$, donde n es el número de elementos que se están ordenando. Esto ocurre cuando la lista se encuentra ordenada en orden inverso, ya que, en cada iteración del algoritmo, se debe comparar y mover cada elemento hacia su posición correcta en la lista, lo que requiere un número cuadrático de operaciones.

En términos de comparaciones y movimientos de elementos, el algoritmo de ordenamiento por inserción en el peor de los casos realiza aproximadamente $\frac{n(n+1)}{2}$ $comparaciones\ y\ \frac{n(n+1)}{2}$ $movimientos\ o\ intercambio\ de\ elementos$:

$$Suma\ de\ Gauss: \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Por lo tanto, para grandes conjuntos de datos, el tiempo de ejecución del algoritmo puede ser significativamente mayor que otros algoritmos de ordenamiento más eficientes, como el ordenamiento por mezcla o el ordenamiento rápido.

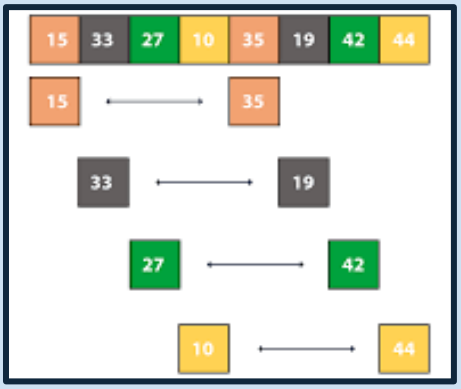
Análisis de complejidad en el mejor de los casos para el ordenamiento por inserción

Esto ocurre cuando la lista ya está ordenada o casi ordenada, ya que, en cada iteración del algoritmo, solo se necesita comparar cada elemento con su predecesor y no se requiere ningún intercambio, lo que resulta en una complejidad de $O(n)$.

Ordenamiento Shell

El método de ordenamiento Shell consiste en dividir el arreglo (o la lista de elementos) en intervalos (o bloques) de varios elementos para organizarlos después por medio del ordenamiento de inserción directa. El proceso se repite, pero con intervalos cada vez más pequeños, de tal manera que al final, el ordenamiento se haga en un intervalo de una sola posición, similar al ordenamiento por inserción directa, la diferencia entre ambos es qué, al final, en el método shell los elementos están ya casi ordenados. [2]

En la siguiente imagen se puede mostrar que el intervalo al que se realizan los cambios es de 4, por lo que podría decirse que el arreglo principal se divide en pequeños subarreglos que van a comparar el dato inicial con el que se encuentra 4 posiciones después, para verificar si es mayor a él y que de esta forma se realice el intercambio.



Tomando en cuenta que cada uno de los intervalos se obtiene al dividir el tamaño del arreglo entre 2, en el peor de los casos, la complejidad para este algoritmo es de O(n2), ya que menor se encuentra un poco después o antes de la mitad, el algoritmo se ejecutará varias veces para poder colocarlo en la primera posición. En el mejor de los casos podría llegar a ser $O(nlog(n))$ya que el arreglo estaría casi ordenado y los intervalos que se realizarían lograrían ordenarlo de forma rápida.

Características del ordenamiento shell

- Se trata de un algoritmo de ordenación interna.
- Se basa en comparaciones e intercambios.
- Necesita que el tiempo de acceso a cualquier dato sea constante.
- En cierto modo, puede considerarse una ampliación del algoritmo de inserción.

Ventajas del ordenamiento shell

- Es fácil de implementar y comprender.
- Es más eficiente que el algoritmo de ordenamiento por inserción, especialmente en listas grandes.
- Puede ser más rápido que otros algoritmos de ordenamiento de complejidad similar.
- No requiere de memoria adicional para su ejecución, ya que el ordenamiento se realiza en la lista original.

Desventajas del ordenamiento shell

- No siempre garantiza el ordenamiento óptimo, lo que significa que puede haber casos en los que la lista no se ordena de la manera más eficiente posible.
- La complejidad del algoritmo depende en gran medida de la secuencia de incrementos elegida, y la selección de una secuencia óptima puede ser difícil.
- El algoritmo es sensible al tamaño del conjunto de datos y la distribución de los elementos en la lista, lo que significa que puede no ser tan efectivo en ciertos casos como en otros algoritmos de ordenamiento.
- No es adecuado para listas muy grandes o para conjuntos de datos que cambian con frecuencia, ya que la reordenación completa de la lista puede ser costosa en términos de tiempo de ejecución.

En resumen, el algoritmo de ordenamiento Shell es una buena opción para conjuntos de datos moderados y uniformemente distribuidos, pero puede no ser la mejor opción para casos extremos o para listas que cambian con frecuencia.

Análisis de complejidad en el peor de los casos para el ordenamiento shell

La complejidad del algoritmo de ordenamiento Shell en el peor de los casos depende de la secuencia de incrementos utilizada. En general, la complejidad del algoritmo de ordenamiento Shell en el peor de los casos se ha estimado en O(n^2), donde n es el número de elementos en la lista que se está ordenando.

Esta complejidad puede ocurrir cuando la secuencia de incrementos elegida para el algoritmo de Shell no es óptima y no puede reducir efectivamente la cantidad de comparaciones y movimientos de elementos necesarios. En tal caso, el algoritmo se comporta como un algoritmo de ordenamiento por inserción básico y puede requerir una cantidad significativa de tiempo para ordenar la lista, especialmente si el tamaño de la lista es grande.

Análisis de complejidad en el peor de los casos para el ordenamiento shell

La complejidad del algoritmo de ordenamiento Shell en el mejor de los casos depende también de la secuencia de incrementos utilizada. La secuencia de incrementos elegida puede reducir significativamente la cantidad de comparaciones y movimientos de elementos necesarios. Cuando se utiliza una secuencia óptima, el algoritmo de ordenamiento Shell puede ordenar una lista de tamaño n en tiempo $O(nlog(n))$, lo cual lo hace bastante eficiente en términos de tiempo de ejecución.

En conclusión, el algoritmo de ordenamiento Shell es un algoritmo de ordenamiento eficiente y ampliamente utilizado en la práctica. Si bien su complejidad en el peor de los casos puede ser $O(n^2)$, la elección de una secuencia óptima puede mejorar significativamente su rendimiento, incluso en comparación con otros algoritmos de ordenamiento de complejidad similar, logrando llegar a una complejidad de $O(nlog(n))$.

Ordenamiento por mezcla

El algoritmo Merge Sort o mezcla es un algoritmo de ordenamiento que sigue un enfoque de "divide y vencerás". En este proceso, se divide la lista de elementos en mitades hasta que se llega a sublistas de un solo elemento, luego se fusionan estas sublistas en pares ordenados, y así sucesivamente hasta que se obtiene una lista ordenada completa.

El algoritmo Merge Sort es eficiente en términos de tiempo de ejecución, ya que su complejidad es O(n log n). Esto lo hace especialmente útil para ordenar grandes conjuntos de datos.

Además, una de las ventajas de Merge Sort es que es un algoritmo estable, lo que significa que mantiene el orden relativo de elementos iguales en la lista original.

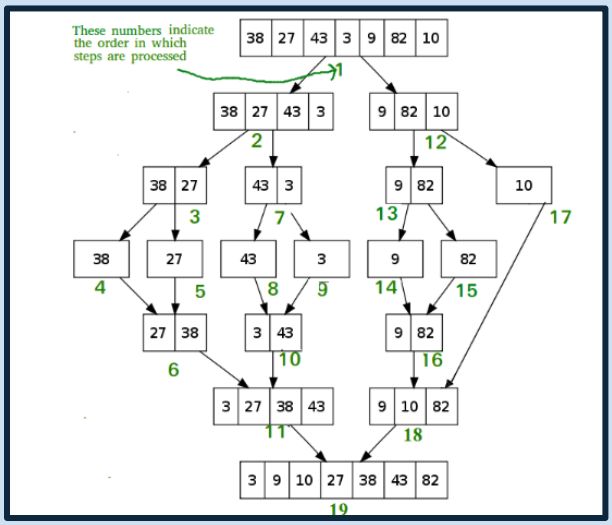
En resumen, Merge Sort es un método de ordenamiento muy efectivo y estable que se basa en la técnica de "divide y conquista". Su complejidad en términos de tiempo de ejecución lo hace adecuado para trabajar con grandes conjuntos de datos.

Fue desarrollada en 1945 por John Von Neumann.

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

1. Si la longitud de la lista es 0 a 1, entonces ya está ordenada, en otro caso.
2. Dividir la lista ordenada en dos sublista de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

La idea principal es que el algoritmo puede mejorar el tiempo de ejecución ya que una lista pequeña necesitará menos pasos para ordenarse que una lista grande.



Ventajas del ordenamiento por mezcla

- Complejidad de tiempo garantizada: El algoritmo de ordenamiento por mezcla tiene una complejidad de tiempo garantizada de O(n log n) tanto en el peor como en el mejor de los casos, lo que lo hace eficiente en términos de tiempo de ejecución para grandes conjuntos de datos.
- Eficiencia en memoria: Aunque el algoritmo de ordenamiento por mezcla necesita una cantidad adicional de memoria para almacenar temporalmente los datos durante el proceso de mezcla, esta cantidad es relativamente pequeña y no aumenta a medida que aumenta el tamaño de los datos, por lo que es más eficiente en memoria que otros algoritmos de ordenamiento.

- Estabilidad: El algoritmo de ordenamiento por mezcla es estable, lo que significa que mantiene el orden relativo de los elementos con valores iguales en la lista de entrada.

Desventajas

- Uso de memoria adicional: Aunque el algoritmo de ordenamiento por mezcla es eficiente en memoria, aún requiere una cantidad adicional de memoria para almacenar temporalmente los datos durante el proceso de mezcla, lo que puede ser un problema en sistemas con recursos limitados.
- No es adecuado para datos pequeños: El algoritmo de ordenamiento por mezcla no es muy eficiente para datos pequeños, ya que la cantidad adicional de memoria necesaria y la sobrecarga del proceso de división recursiva pueden hacer que sea más lento que otros algoritmos de ordenamiento más simples como el de ordenamiento por inserción.
- No es in-situ: Es decir, no ordena los elementos en la lista original, sino que crea una nueva lista ordenada y por lo tanto requiere de memoria adicional. Esto puede ser un problema en situaciones en las que la memoria es limitada.

Aplicaciones del ordenamiento por mezcla

- Clasificación de grandes conjuntos de datos: la clasificación por combinación es particularmente adecuada para clasificar grandes conjuntos de datos debido a su complejidad de tiempo garantizada en el peor de los casos de O (n log n). Esto lo convierte en una opción popular para ordenar algoritmos utilizados en bases de datos y otras aplicaciones de uso intensivo de datos.
- Clasificación externa: la clasificación por combinación se usa comúnmente en la clasificación externa, donde los datos a clasificar son demasiado grandes para caber en la memoria. La ordenación por combinación se puede adaptar para que funcione con dispositivos de almacenamiento externo, como discos duros o unidades de cinta, lo que la hace útil para aplicaciones como la clasificación de archivos grandes o el procesamiento de flujos de datos.
- Procesamiento paralelo: Merge sort es un algoritmo naturalmente paralelizable, lo que significa que se puede adaptar fácilmente para trabajar con múltiples procesadores o subprocesos. Esto lo hace útil para aplicaciones que requieren computación de alto rendimiento, como simulaciones científicas o modelos financieros.
- Ordenación estable: la ordenación por combinación es un algoritmo de ordenación estable, lo que significa que mantiene el orden relativo de elementos iguales en la matriz de entrada. Esto lo hace útil en aplicaciones donde es importante preservar el orden original de elementos iguales, como en bases de datos o sistemas financieros.
- Ordenación personalizada: la ordenación combinada se puede adaptar para manejar diferentes distribuciones de entrada, como datos parcialmente ordenados, casi ordenados o completamente sin ordenar. Esto lo hace útil en una variedad de aplicaciones del mundo real, donde los datos pueden tener distribuciones complejas y variadas.

Análisis de complejidad en el peor de los casos para el ordenamiento de mezcla

El peor de los casos para el algoritmo de ordenamiento Merge se produce cuando la lista está completamente desordenada y cada elemento debe ser comparado y movido en la

fase de combinación. En esta situación, el algoritmo de ordenamiento Merge dividirá la lista en dos mitades iguales y cada mitad se ordenará recursivamente. Cada una de estas mitades tendrá $\frac{n}{2}$ elementos, y el proceso de ordenación recursiva continuará hasta que se llegue a una lista con un solo elemento.

Luego, se realizará la fase de combinación, que requiere que se compare cada uno de los n elementos de ambas mitades para determinar en qué orden deben colocarse en la lista resultante, en el peor de los casos se necesitarán comparaciones para cada uno de los n elementos, lo que resulta en una complejidad de $O(n)$. Debido a que la lista se divide en dos mitades iguales en cada llamada recursiva, el número de llamadas recursivas es $O(\log(n))$. Por lo tanto, la complejidad resultante o total del algoritmo de ordenamiento Merge en el peor de los casos es $O(nlog(n))$.

Análisis de complejidad en el mejor de los casos para el ordenamiento de mezcla

El mejor de los casos para el algoritmo de ordenamiento Merge se produce cuando la lista ya se encuentra completamente ordenada. En este caso, el algoritmo simplemente realiza una comparación entre cada elemento de ambas mitades y los combina en orden, sin necesidad de realizar ninguna otra operación, llegando a una complejidad de $O(nlog(n))$, igual que en el peor de los casos, sin embargo, es importante destacar que, aunque la complejidad de tiempo en el mejor de los casos es la misma que en el peor de los casos, la velocidad de ejecución del algoritmo será significativamente más rápida, ya que se necesitarán muchas menos operaciones.

En general, el algoritmo de ordenamiento Merge es una buena opción para una amplia gama de situaciones de ordenamiento, tanto en el mejor como en el peor de los casos.

Espacio auxiliar para el algoritmo por mezcla

El algoritmo de mezcla (Merge Sort) utiliza una cantidad adicional de memoria para fusionar los subconjuntos ordenados en el proceso de mezcla, por lo tanto, su espacio auxiliar o memoria es de $O(n)$ en el peor de los casos.

Ordenamiento rápido

El ordenamiento rápido, también conocido como QuickSort, es un algoritmo de ordenamiento que utiliza un enfoque de "divide y conquista". El proceso comienza seleccionando un elemento, conocido como pivote, de la lista a ordenar y luego se divide la lista en dos subconjuntos: uno con elementos más pequeños que el pivote y otro con elementos más grandes.

Luego, se aplica el mismo proceso de partición a cada uno de los subconjuntos, seleccionando un nuevo pivote y dividiendo cada subconjunto en dos sublistas, y así sucesivamente hasta que todos los elementos estén en su posición ordenada.

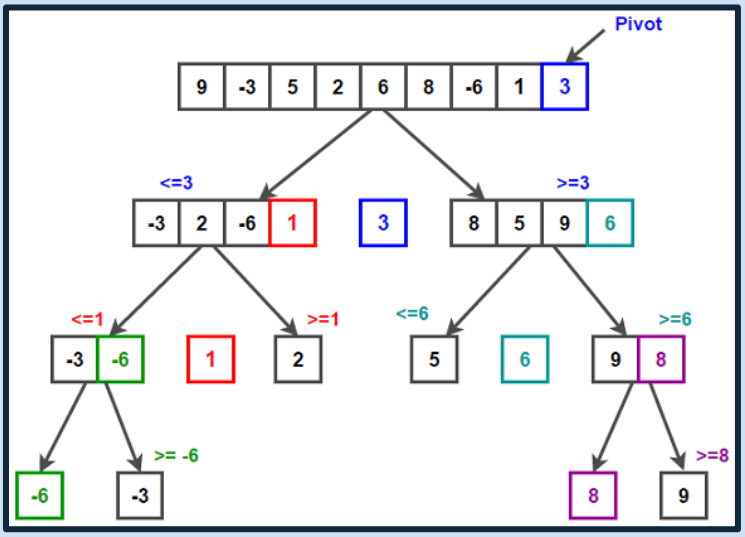
El algoritmo QuickSort es muy eficiente en términos de tiempo de ejecución, ya que su complejidad promedio es de O(n log n). Sin embargo, su peor caso puede tener una complejidad de $O(n^2)$, lo que lo hace menos adecuado para ordenar conjuntos de datos muy grandes.

Una de las ventajas de QuickSort es que se puede implementar de manera recursiva, lo que simplifica el código y lo hace más fácil de entender y mantener. Además, el ordenamiento rápido no requiere espacio adicional para almacenar los datos, lo que lo hace más eficiente en términos de memoria que otros algoritmos de ordenamiento.

El ordenamiento rápido también conocido como QuickSort, es un algoritmo de ordenamiento eficiente y popular utilizado para ordenar listas de elementos. El algoritmo se basa en la técnica de divide y vencerás.

Pasos del ordenamiento rápido

1. Selección en un pivote: Se selecciona un elemento arbitrario de la lista, llamado pivote.
2. Partición de la lista: Se divide en dos sublistas, una contiene los elementos menores que el pivote y otra que contiene los elementos mayores que el pivote. Para lograrlo,se recorre la lista y se comparacon cada elemento con el pivote. Si el elemento es menor, se coloca en la sublista izquierda, y si el elemnto es mayor, en la sublista derecha.
3. Recursión: se aplica el algoritmo de ordenamiento rápido a cada una de las sublistas generadas en el paso anterior. Este proceso se repite hasta que todas las sublistas contengan solo un elemento, en cuyo caso la lista está completamente ordenada.



Ventajas del ordenamiento rápido

- Complejidad de tiempo promedio rápida: El algoritmo de ordenamiento rápido tiene una complejidad de tiempo promedio de O(n log n), lo que lo hace muy rápido para grandes conjuntos de datos.
- Uso eficiente de memoria: El algoritmo de ordenamiento rápido es in-place, lo que significa que no requiere memoria adicional para realizar la ordenación. Esto lo hace muy eficiente en términos de memoria para grandes conjuntos de datos.
- Implementación sencilla: El algoritmo de ordenamiento rápido es relativamente fácil de implementar y comprender en comparación con otros algoritmos de ordenamiento como el de ordenamiento por mezcla.
- Buen desempeño en conjuntos de datos aleatorios: El algoritmo de ordenamiento rápido tiene un buen desempeño en conjuntos de datos aleatorios, ya que es probable que el

conjunto de datos se divida de manera uniforme y se eviten las peores situaciones de división.
- Bueno para ordenar datos repetidos: El algoritmo de ordenamiento rápido es bueno para ordenar conjuntos de datos que contienen valores repetidos, ya que puede evitar la sobrecarga en el proceso de división que se produce en otros algoritmos como el de ordenamiento por mezcla.

En general, el algoritmo de ordenamiento rápido es una excelente opción para ordenar grandes conjuntos de datos, especialmente si se dispone de memoria limitada y se requiere una complejidad de tiempo rápida.

Desventajas del ordenamiento rápido

- Complejidad de tiempo en el peor de los casos: En el peor de los casos, el algoritmo de ordenamiento rápido puede tener una complejidad de tiempo de $O(n^2)$, lo que lo hace ineficiente en comparación con otros algoritmos como el de ordenamiento por mezcla, pero la probabilidad de que esto ocurra en la práctica es muy baja, sin embargo, es importante tenerlo en cuenta.
- No es estable: El algoritmo de ordenamiento rápido no es un algoritmo de ordenamiento estable, lo que significa que no garantiza el orden relativo de elementos con claves iguales.
- Dependencia de la elección del pivote: El algoritmo de ordenamiento rápido depende en gran medida de la elección del pivote, lo que puede afectar significativamente su desempeño. Si el pivote se elige mal, el algoritmo puede tomar mucho más tiempo para ordenar el conjunto de datos.
- No es adecuado para ordenar datos sensibles a comparaciones: El algoritmo de ordenamiento rápido no es adecuado para ordenar conjuntos de datos que son sensibles a comparaciones, como las cadenas de texto, ya que las comparaciones de elementos pueden ser muy costosas en términos de tiempo.

En general, el algoritmo de ordenamiento rápido es una buena opción para la mayoría de los casos, especialmente cuando se trata de grandes conjuntos de datos y se dispone de memoria limitada. Sin embargo, es importante tener en cuenta sus limitaciones y considerar otros algoritmos de ordenamiento si se requiere una estabilidad o si los datos a ordenar son sensibles a comparaciones.

Análisis de complejidad en el mejor de los casos para el ordenamiento rápido

Esto ocurre cuando el pivote elegido divide el conjunto de datos en dos subconjuntos de tamaño igual (o muy cercano). En este caso, el algoritmo divide recursivamente el conjunto de datos en dos mitades y aplica el mismo proceso en cada una de ellas. Como resultado, el número de comparaciones necesarias para ordenar el conjunto de datos es proporcional a su tamaño multiplicado por el logaritmo de su tamaño, lo que da como resultado una complejidad de tiempo de O(n log n) en el mejor de los casos.

Además, en el mejor de los casos, el algoritmo de ordenamiento rápido tiene una complejidad de espacio de O(log n), ya que utiliza la recursividad para dividir el conjunto de datos en subconjuntos más pequeños.

Es importante destacar que la elección del pivote puede afectar significativamente el rendimiento del algoritmo en el mejor de los casos. Una buena elección de pivote puede garantizar que el conjunto de datos se divida en dos subconjuntos de tamaño igual o muy cercano, lo que resulta en una complejidad de tiempo óptima. Sin embargo, una mala elección de pivote puede hacer que el algoritmo de ordenamiento rápido sea menos eficiente en el mejor de los casos, acercándose a la complejidad de tiempo en el peor de los casos.

Análisis de complejidad en el peor de los casos para el ordenamiento rápido

Esto ocurre cuando el pivote elegido divide el conjunto de datos en subconjuntos desequilibrados, es decir, uno de los subconjuntos resulta ser mucho más grande que el otro. En este caso, el algoritmo no puede dividir recursivamente el conjunto de datos en dos mitades de tamaño similar, lo que resulta en una complejidad de tiempo cuadrática.

Por ejemplo, si el pivote elegido siempre es el elemento más pequeño o grande del conjunto de datos, el algoritmo no podrá dividir el conjunto en dos subconjuntos de tamaño similar, lo que resultará en el peor de los casos.

Sin embargo, en promedio, el algoritmo de ordenamiento rápido tiene una complejidad de tiempo de O(n log n), lo que lo hace muy eficiente para la mayoría de los conjuntos de datos. La elección adecuada del pivote y la aplicación de técnicas de optimización, como la eliminación de la recursividad de cola, pueden mejorar aún más el rendimiento del algoritmo.

Espacio auxiliar para el algoritmo rápido

A diferencia del algoritmo de mezcla, el algoritmo de ordenamiento rápido no utiliza memoria adicional para fusionar subconjuntos, realiza los intercambios in-situ, es decir, sobre la misma colección de elementos. Sin embargo, el algoritmo de ordenamiento rápido requiere memoria adicional para mantener la pila de llamadas de la función de partición, por lo que la cantidad de memoria adicional requerida depende de la profundidad máxima de la pila de llamadas. En el peor de los casos, cuando el conjunto de datos está invertido, la profundidad máxima de la pila de llamadas y espacio auxiliar es igual a $O(n)$. En el mejor de los casos, cuando el conjunto de datos está ordenado o casi ordenado, la profundidad máxima de la pila de llamadas y espacio auxiliar es igual a $O(\log(n))$.

Es importante tener en cuenta que la complejidad de espacio adicional del algoritmo de ordenamiento rápido puede variar según la implementación específica y las decisiones de diseño. En general, el algoritmo de ordenamiento rápido es eficiente en términos de tiempo y espacio cuando se implementa correctamente.

Planteamiento del problema

En esta práctica se busca encontrar el mejor algoritmo de ordenamiento, dado su tiempo de ejecución para diferentes cantidades de datos que se ingresarán. Para esta práctica se tienen que implementar los algoritmos de ordenamiento (burbuja, inserción, selección, shell, mezcla y rápido), los cuales debían realizarse en C. Una vez hecho esto, se medirán los tiempos de ejecución de cada algoritmo, probándolo con una cantidad diferente de n

elementos leídos de un archivo con un millón de números proporcionado por el profesor. Para poder realizar todo esto, se tuvieron que analizar los diferentes pseudocódigos de los ordenamientos, para poder tener una referencia de cómo es que se ejecutaba el código y posteriormente ponerlo en práctica. Todo esto se realizó con el fin de comprender su funcionamiento y determinar su tiempo de ejecución, ya que dadas las diferentes cantidades de elementos ingresados como parámetro para que fueran ordenados, se realizó una comparativa de sus tiempos.

Diseño y funcionamiento de la solución

A continuación, se encuentran los pseudocódigos y describen los algoritmos de ordenamiento empleados en la práctica:

Algoritmo de ordenamiento de burbuja optimizada

```
Algoritmo BurbujaOptimizada(A, n)
    cambios = SI
    i = 0
    mientras i<=n-2 && cambios != NO hacer
        cambios = NO
        para j=0 hasta (n-2)-i hacer
            Si (A[j] < A[j+1]) hacer
                aux = A[j]
                A[j] = A[j+1]
                A[j+1] = aux
                cambios = SI
            fin si
        fin para
        i = i + 1
    fin mientras
fin algoritmo
```

Pseudocódigo de burbuja optimizada

Burbuja: Este algoritmo compara los elementos adyacentes de la lista y los intercambia si no están en orden. Continúa haciendo esto a través de la lista hasta que todos los elementos estén ordenados. El nombre "burbuja" se debe a que los elementos más grandes "flotan" hacia la parte superior de la lista, como si fueran burbujas de aire.

Explicación del algoritmo de Burbuja Optimizada

1. Inicializar la variable 'cambios' a 'SI' y la variable 'i' a 0.
2. Mientras 'i' sea menor o igual a 'n-2' y 'cambios' sea diferente de 'NO', hacer lo siguiente:
    a. Inicializar la variable 'cambios' a 'NO'.
    b. Iniciar un ciclo 'para' que recorra desde 0 hasta 'n-2-i'.
    c. Si el elemento en la posición 'j' es menor que el elemento en la posición 'j+1', intercambiarlos y marcar la variable 'cambios' como 'SI'.

   d. Finalizar el ciclo 'para'.
   e. Incrementar la variable 'i' en 1.
3. Finalizar el ciclo 'mientras'.
4. Fin del algoritmo.

Algoritmo de ordenamiento por inserción

```
Algoritmo Insercion(A, n)
    para i=0 hasta n-1 hacer
        j = i
        temp = A[i]
        mientras (j > 0) && (temp < A[j-1]) hacer
            A[j] = A[j-1]
            j--
        fin mientras
        A[j] = temp
    fin para
fin Algoritmo
```

Pseudocódigo de ordenamiento por inserción

Inserción: En este algoritmo, los elementos se insertan en la lista en orden. El primer elemento se considera ordenado, y se compara el siguiente elemento con él para determinar dónde debe ser insertado. Este proceso continúa hasta que todos los elementos estén en su posición correcta.

Explicación del algoritmo de Insercion

1. Recibe un arreglo A de tamaño n como entrada.
2. Para cada elemento en el arreglo A, comienza un ciclo for que inicia en i=0 y termina en n-1.
3. Dentro del ciclo for, se define una variable j=i, y se guarda el valor del elemento actual en una variable temporal llamada temp.
4. Mientras j sea mayor que 0 y el valor temporal sea menor que el valor en la posición anterior del arreglo (A[j-1]), se entra en un ciclo while.
5. Dentro del ciclo while, el valor en la posición actual de A (A[j]) se actualiza con el valor en la posición anterior (A[j-1]), y se decrementa j en 1.
6. Una vez que se ha terminado el ciclo while, se actualiza el valor en la posición j de A con el valor temporal.
7. Se repite el ciclo for para el siguiente elemento en el arreglo.
8. El arreglo A ha sido ordenado en orden ascendente.

Algoritmo de ordenamiento por selección

```
Algoritmo Seleccion(A, n)
    para k=0 hasta n-2 hacer
        p=k
        para i=k+1 hasta n-1 hacer
            si A[i] < A[p] entonces
                p=i
            fin si
        fin para
        temp = A[p]
        A[p] = A[k]
        A[k] = temp
    fin para
fin Algoritmo
```

Pseudocódigo de ordenamiento por selección

Selección: En este algoritmo, se busca el elemento más pequeño en la lista y se coloca en la primera posición. Luego, se busca el siguiente elemento más pequeño y se coloca en la segunda posición, y así sucesivamente hasta que todos los elementos estén ordenados.

Explicación del algoritmo de Seleccion

1. Se define una función llamada Seleccion que toma un arreglo A de tamaño n como entrada.
2. Se inicia un bucle para recorrer el arreglo desde la posición k=0 hasta n-2.
3. Se asigna la variable p=k, que se utiliza para encontrar el índice del elemento más pequeño en el arreglo.
4. Se inicia otro bucle anidado que recorre desde k+1 hasta n-1.
5. Si el elemento en la posición i es menor que el elemento en la posición p, se actualiza el valor de p a i.
6. Al final del bucle anidado, se intercambia el elemento en la posición p con el elemento en la posición k.
7. El bucle principal se repite para la siguiente posición del arreglo, k+1.
8. Cuando el bucle principal termina, el arreglo está ordenado de forma ascendente.
9. Fin del algoritmo.

Algoritmo de ordenamiento shell

```
Algoritmo Shell(A, n)
    k = TRUNC(n / 2)
    mientras k >= 1 hacer
        b = 1
        mientras b != 0 hacer
            b = 0
            para i=k hasta n-1 hacer
                si A[i-k] > A[i]
                    temp = A[i]
                    A[i] = A[i-k]
                    A[i-k] = temp
                    b = b + 1
                fin si
            fin para
        fin mientras
        k = TRUNC(k / 2)
    fin mientras
fin algoritmo
```

Pseudocódigo de ordenamiento shell

Shell: Este algoritmo es una versión mejorada del método de inserción. Divide la lista en subgrupos más pequeños, los ordena utilizando el método de inserción y luego los combina para producir una lista ordenada. La idea es reducir el número de comparaciones necesarias para ordenar la lista.

Explicación del algoritmo Shell

1. Se inicia definiendo el tamaño de la brecha inicial k como la mitad del tamaño del arreglo: k = TRUNC(n / 2).
2. Se entra en un bucle while que se ejecutará mientras k sea mayor o igual a 1.
3. Dentro del bucle while, se establece una variable b en 1, que se usará para indicar si se han realizado intercambios o no.
4. Se inicia un bucle for que recorre el arreglo a partir de la posición k: para i=k hasta n-1 hacer.
5. Dentro del bucle for, se verifica si el elemento en la posición i-k es mayor que el elemento en la posición i.
6. Si el elemento en la posición i-k es mayor que el elemento en la posición i, se intercambian los elementos.
7. Se incrementa la variable b en 1 para indicar que se ha realizado un intercambio.
8. Se cierra el bucle for.
9. Si la variable b es igual a cero, significa que no se han realizado intercambios, por lo que se sale del bucle while.
10. Si la variable b es distinta de cero, se divide k por 2 y se trunca el resultado: k = TRUNC(k / 2).

11. Se repite el bucle while desde el paso 2 hasta que k sea menor que 1.
12. Fin del algoritmo.

Algoritmo de ordenamiento de mezcla

```
Algoritmo MergeSort(A, p, r)
    si (p < r) entonces
        q = parteEntera((p + r) / 2)
        MergeSort(A, p, q)
        MergeSort(A, q+1, r)
        Merge(A, p, q, r)
    fin si
fin Algoritmo

Algoritmo Merge(A, p, q, r)
    l=r-p+1, i=p, j=q+1
    *C = malloc(l * sizeof(int))
    para k=0 hasta l-1 hacer
        si (i <= q && j <= r)
            si (A[i] < A[j])
                C[k] = A[i]
                i++
            sino
                C[k] = A[j]
                j++
            fin si
        sino si (i <= q)
            C[k] = A[i]
            i++
        sino
            C[k] = A[j]
            j++
        fin si
    fin para
    k = p
    para i=0 hasta l-1 hacer
        A[k] = C[i]
        k++
    fin para
    free(C)
fin Algoritmo
```

Pseudocódigo de algoritmo de mezcla

Mezcla: Este algoritmo divide la lista en subgrupos más pequeños, ordena cada subgrupo y luego los combina en una sola lista ordenada. Este proceso se repite hasta que todos los subgrupos estén combinados en una sola lista ordenada.

Explicación del algoritmo MergeSort

1. Si la posición inicial p es menor que la posición final r, se ejecuta el algoritmo.
2. Se calcula la posición media q como la parte entera de la suma de p y r dividida entre 2.
3. Se llama a MergeSort con los mismos valores iniciales de p y q para la primera mitad de la lista.
4. Se llama a MergeSort con los valores de q+1 y r para la segunda mitad de la lista.
5. Se llama a Merge para combinar ambas mitades en una sola lista ordenada.

Explicación del algoritmo Merge

1. Se calcula la longitud l de la sublista que se va a combinar y se inicializan las variables i y j a las posiciones iniciales de ambas sublistas.
2. Se crea una nueva lista C del tamaño de la longitud l.
3. Se recorre la lista C utilizando una variable k como índice y se van comparando los elementos de las dos sublistas utilizando los índices i y j. El elemento más pequeño se agrega a la lista C y se incrementa el índice correspondiente.
4. Si alguna sublista tiene elementos sobrantes, se copian en la lista C.
5. Se copia la lista C de regreso a la lista original A.
6. Se libera la memoria de la lista C.

Algoritmo de ordenamiento rápido

```
Algoritmo Quicksort(A, p, r)
    si p < r entonces
        j = Pivot(A, p, r)
        QuickSort(A, p, j-1)
        QuickSort(A, j+1, r)
    fin si
fin Algoritmo

Algoritmo Pivot(A, p, r)
    piv = A[p], i = p + 1, j = r
    mientras (i <= j)
        mientras A[i] <= piv && i <= r hacer
            i++
        mientras A[j] > piv hacer
            j--
        si i < j entonces
            Intercambiar(A, i, j)
    fin mientras
    Intercambiar(A, p, j)
    regresar j
```

```
fin Algoritmo

Algoritmo Intercambiar(A, i, j)
    temp = A[j]
    A[j] = A[i]
    A[i] = temp
fin Algoritmo
```

Pseudocódigo de algoritmo rápido

Rápido: En este algoritmo, se selecciona un elemento de la lista como "pivote", y los elementos restantes se dividen en dos grupos: aquellos mayores que el pivote y aquellos menores que el pivote. Luego, los dos grupos se ordenan por separado y se combinan para producir una lista ordenada. Este proceso se repite recursivamente hasta que toda la lista esté ordenada.

Explicación del algoritmo de quicksort

1. Se recibe como parámetros el arreglo A, y los índices p y r, que representan la posición del primer y último elemento del subarreglo a ordenar.
2. Se verifica que p sea menor que r, si es así, entonces:
3. Se llama a la función Pivot, que devuelve el índice j del elemento pivote que divide al arreglo en dos subarreglos: los elementos menores al pivote a la izquierda y los mayores a la derecha.
4. Se llama recursivamente a QuickSort para ordenar el subarreglo de la izquierda, desde la posición p hasta j-1.
5. Se llama recursivamente a QuickSort para ordenar el subarreglo de la derecha, desde la posición j+1 hasta r.
6. Si p es mayor o igual que r, entonces no hay más elementos por ordenar y el algoritmo termina.
7. Fin del algoritmo.

Explicación del algoritmo Pivot

1. Se recibe como parámetros el arreglo A, y los índices p y r, que representan la posición del primer y último elemento del subarreglo a ordenar.
2. Se selecciona el elemento A[p] como pivote, y se inicializan los índices i y j en p+1 y r respectivamente.
3. Se entra en un bucle mientras i sea menor o igual que j:
4. Se busca la posición i tal que A[i] sea mayor que el pivote.
5. Se busca la posición j tal que A[j] sea menor o igual que el pivote.
6. Si i es menor que j, se intercambian los elementos A[i] y A[j].
7. Si i es mayor o igual que j, se sale del bucle.
8. Se intercambia el pivote A[p] con A[j], para colocar el pivote en su posición final en el arreglo.
9. La función Pivot regresa el índice j.
10. Fin del algoritmo.

Explicación del algoritmo intercambiar

1. Se recibe como parámetros el arreglo A, y los índices i y j que representan las posiciones de los elementos a intercambiar.
2. Se guarda el valor del elemento A[j] en una variable temporal temp.
3. Se asigna el valor de A[i] a A[j].
4. Se asigna el valor de temp a A[i].
5. La función Intercambiar termina.
6. Fin del algoritmo.

Complejidad de tiempo y espacio de los algoritmos de ordenamiento

- Burbuja optimizada: Tiempo - $O(n^2)$, Espacio - $O(1)$
- Inserción: Tiempo - $O(n^2)$, Espacio - $O(1)$
- Selección: Tiempo - $O(n^2)$, Espacio - $O(1)$
- Mezcla: Tiempo - $O(nlog(n))$, Espacio - O(n)
- Rápido: Tiempo - $O(nlog(n))$ promedio, $O(n^2)$ en el peor de los casos, Espacio - $O(\log(n))$
- Shell: Tiempo - depende de la secuencia utilizada, pero puede ser tan bueno como $O(n\ log(n))$, Espacio - $O(1)$

Implementación de la solución

Implementación del algoritmo de ordenamiento de burbuja optimizada:

```
void BurbujaOptimizada(int *A, int n){
    int cambios=TRUE;
    int i=0;
    while(i<=n-2 && cambios != FALSE){
        cambios=FALSE;
        for(int j=0; j<=(n-2)-i; j++){
            if(A[j] > A[j+1]){
                int aux=A[j];
                A[j]=A[j+1];
                A[j+1]=aux;
                cambios=TRUE;
            }
        }
        i=i+1;
    }
}
```

Algoritmo de burbuja optimizada

Explicación del algoritmo de burbuja optimizada

1. La función BurbujaOptimizada toma un arreglo A y el tamaño del arreglo n como parámetros.
2. Se inicializa una variable llamada cambios como TRUE y un índice i como 0. El índice i se utiliza para llevar la cuenta de cuántas veces se ha recorrido la lista.
3. Se utiliza un ciclo while para recorrer la lista hasta que se haya completado una pasada completa de la lista y no haya habido cambios en la pasada.
4. Dentro del ciclo while, se inicializa la variable cambios como FALSE. Esto se hace para comprobar si se ha realizado algún cambio en la pasada actual. Si no se realiza ningún cambio, se detiene el proceso de ordenamiento.
5. Se utiliza otro ciclo for para recorrer la lista. El índice j se utiliza para llevar la cuenta de la posición actual en la lista.
6. Dentro del ciclo for, se compara el elemento actual A[j] con el siguiente elemento A[j+1]. Si el elemento actual es mayor que el siguiente elemento, se intercambian los elementos. Esto asegura que los elementos mayores se muevan hacia la derecha.
7. Si se intercambian los elementos, se establece cambios como TRUE. Esto significa que se ha realizado un cambio en la pasada actual.
8. Después de recorrer la lista completa con el ciclo for, se aumenta el índice i en 1. Esto indica que se ha completado una pasada completa de la lista.
9. Si el índice i es menor que el tamaño de la lista menos 1 y se ha realizado algún cambio en la pasada actual (es decir, cambios es TRUE), se vuelve al paso 3 para continuar ordenando la lista.
10. Si el índice i es igual al tamaño de la lista menos 1 o no se ha realizado ningún cambio en la pasada actual, se detiene el proceso de ordenamiento y la lista está ordenada.

Implementación del algoritmo de ordenamiento por inserción

```c
void Insercion(int *A, int n) {
    for (int i = 0; i <= n - 1; i++) {
        int j = i;
        int temp = A[i];
        while (j > 0 && temp < A[j - 1]) {
            A[j] = A[j - 1];
            j--;
        }
        A[j] = temp;
    }
}
```

Algoritmo de ordenamiento por inserción

Explicación del algoritmo de ordenamiento por inserción

1. La función Insercion toma un arreglo A y el tamaño del arreglo n como parámetros.

2. Se utiliza un ciclo for para recorrer el arreglo A, con una variable i que representa la posición actual del elemento que se está insertando.
3. Se inicializa la variable j como i, que representa la posición actual del elemento que se está comparando.
4. Se almacena el valor del elemento actual en una variable temporal llamada temp.
5. Se utiliza un ciclo while para comparar el valor del elemento actual con los elementos anteriores a él en la lista ordenada. Si el valor de temp es menor que el elemento anterior A[j-1], entonces se desplaza el elemento anterior hacia la derecha y se decrementa la variable j.
6. Se repite el paso 5 hasta que se encuentre un elemento anterior que sea menor o igual que el valor de temp, o hasta que se llegue al inicio de la lista.
7. Se inserta el valor de temp en la posición correcta en la lista ordenada, que es la posición j.
8. Se repite este proceso hasta que se hayan insertado todos los elementos en la lista.

Implementación del algoritmo de ordenamiento por selección

```
void Seleccion(int *A, int n) {
    for (int k = 0; k <= n - 2; k++) {
        int p = k;
        for (int i = k + 1; i <= n - 1; i++) {
            if (A[i] < A[p])
                p = i;
        }
        int temp = A[p];
        A[p] = A[k];
        A[k] = temp;
    }
}
```

Algoritmo de ordenamiento por selección

Explicación del algoritmo por selección:

1. La función Seleccion toma un arreglo A y el tamaño del arreglo n como parámetros.
2. Se utiliza un ciclo for para recorrer el arreglo A, con una variable k que representa la posición actual del elemento más pequeño que no ha sido ordenado.
3. Se inicializa la variable p como k, que representa la posición del elemento más pequeño en el subarreglo no ordenado.
4. Se utiliza otro ciclo for para recorrer el subarreglo no ordenado desde k+1 hasta n-1.
5. Dentro del ciclo for, se compara el elemento actual A[i] con el elemento más pequeño en el subarreglo no ordenado, A[p]. Si el elemento actual es menor que el elemento más pequeño, se actualiza la posición p para apuntar al nuevo elemento más pequeño.

6. Después de recorrer todo el subarreglo no ordenado, se intercambia el elemento más pequeño (A[p]) con el elemento actual (A[k]). Esto garantiza que el elemento más pequeño del subarreglo no ordenado esté en su lugar correcto en la lista ordenada.
7. Se repite este proceso hasta que se hayan ordenado todos los elementos en la lista.

Implementación del algoritmo de ordenamiento Shell

```
void Shell(int *A, int n)
{
    int k = n / 2;
    while (k >= 1)
    {
        int b = 1;
        while (b != 0)
        {
            b = 0;
            for (int i = k; i <= n - 1; i++)
            {
                if (A[i - k] > A[i])
                {
                    int temp = A[i];
                    A[i] = A[i - k];
                    A[i - k] = temp;
                    b = b + 1;
                }
            }
        }
        k = k / 2;
    }
}
```

Algoritmo de ordenamiento shell

Explicación del algoritmo de ordenamiento Shell:

1. Se comienza dividiendo el arreglo de entrada en varios subarreglos más pequeños de longitud "k". El valor inicial de "k" suele ser la mitad del tamaño del arreglo de entrada.
2. Para cada subarreglo, se aplican los pasos del algoritmo de inserción para ordenar los elementos del subarreglo.
3. Se reduce el valor de "k" a la mitad y se repiten los pasos 1 y 2 hasta que "k" sea igual a 1.
4. Finalmente, se aplica el algoritmo de inserción para ordenar el arreglo completo.

Implementación del algoritmo de ordenamiento por mezcla

```
void MergeSort(int *A, int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        Merge(A, p, q, r);
    }
}
void Merge(int *A, int p, int q, int r)
{
    int l = r - p + 1, i = p, j = q + 1;
    int *C = malloc(l * sizeof(int));
    if (C == NULL) {
        printf("No se pudo reservar memoria dinámica\n");
        exit(1);
    }
    for (int k = 0; k < l; k++) {
        if (i <= q && j <= r) {
            if (A[i] < A[j]) {
                C[k] = A[i];
                i++;
            }
            else {
                C[k] = A[j];
                j++;
            }
        }
        else if (i <= q) {
            C[k] = A[i];
            i++;
        }
        else {
            C[k] = A[j];
```

```
            j++;
        }
    }
    int k = p;
    for (int i = 0; i < l; i++) {
        A[k] = C[i];
        k++;
    }
    free(C);
}
```

Algoritmo de ordenamiento por mezcla

Explicación del algoritmo de ordenamiento por mezcla

1. Se llama a la función MergeSort con el arreglo original A, el índice del primer elemento p y el índice del último elemento r.
2. En la función MergeSort, si p es menor que r, se calcula el índice medio q y se llama a MergeSort de manera recursiva para la primera mitad del arreglo (p a q) y para la segunda mitad (q+1 a r).
3. En la función MergeSort, cuando las llamadas recursivas terminan, se llama a la función Merge para combinar y ordenar las dos mitades.
4. En la función Merge, se crea un arreglo temporal C con el tamaño del número de elementos que hay en las dos mitades.
5. Se copian los elementos de las dos mitades ordenadas en los arreglos temporales C1 y C2 en orden ascendente.
6. Se copian los elementos de los arreglos temporales C1 y C2 de vuelta en el arreglo original A en orden ascendente.
7. Se libera la memoria reservada para el arreglo temporal C.

Implementación del algoritmo de ordenamiento rápido

```
void Quicksort(int *A, int p, int r) {
    if (p < r) {
        int j = Pivot(A, p, r);
        Quicksort(A, p, j - 1);
        Quicksort(A, j + 1, r);
    }
}
int Pivot(int *A, int p, int r) {
    int piv = A[p], i = p + 1, j = r;
    while (i <= j) {
```

```
        while (A[i] <= piv && i <= r)
            i++;
        while (A[j] > piv)
            j--;
        if (i < j)
            Intercambiar(A, i, j);
    }
    Intercambiar(A, p, j);
    return j;
}
void Intercambiar(int *A, int i, int j) {
    int temp = A[j];
    A[j] = A[i];
    A[i] = temp;
}
```

Algoritmo de ordenamiento rápido

Explicación del algoritmo de ordenamiento rápido:

1. La función Quicksort toma un arreglo A, un índice inicial p y un índice final r como parámetros. Si p es menor que r, la función continúa con el proceso de ordenamiento.
2. La función Pivot toma un arreglo A, un índice inicial p y un índice final r como parámetros. En esta función, se selecciona el elemento pivote como el primer elemento del arreglo A[p]. Luego, se establecen dos índices i y j. El índice i comienza en p + 1 y el índice j comienza en r.
3. Se realiza un ciclo while para reorganizar el arreglo A de manera que todos los elementos menores que el pivote estén a su izquierda y todos los mayores estén a su derecha. El ciclo while continúa mientras el índice i es menor o igual al índice j.
4. En el ciclo while, se mueve el índice i hacia la derecha mientras A[i] es menor o igual que el pivote y el índice i está dentro del rango permitido (es decir, i <= r).
5. En el ciclo while, se mueve el índice j hacia la izquierda mientras A[j] es mayor que el pivote.
6. Si el índice i es menor que el índice j, se intercambian los elementos A[i] y A[j]. Esto asegura que todos los elementos menores que el pivote estén a su izquierda y todos los mayores estén a su derecha.
7. Después de que el ciclo while ha terminado, se intercambia el pivote A[p] con A[j]. El elemento en la posición j es ahora el pivote y se asegura que todos los elementos menores que el pivote estén a su izquierda y todos los mayores estén a su derecha.
8. La función Pivot devuelve el índice j, que representa la posición del pivote en la lista ordenada.
9. En la función Quicksort, se llama a la función Pivot para obtener el índice j del pivote. Luego, se llama a la función Quicksort dos veces: una vez para ordenar los elementos

menores que el pivote (desde p hasta j-1) y otra vez para ordenar los elementos mayores que el pivote (desde j+1 hasta r).

10. El proceso de ordenamiento se aplica recursivamente a las sub-listas generadas por el pivote hasta que todo el arreglo esté ordenado.

Actividades

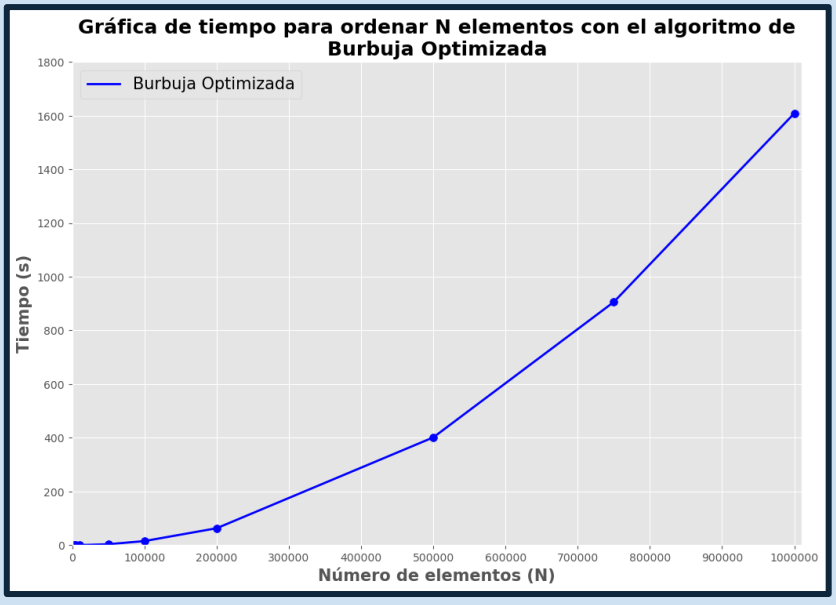
Pruebas de tiempo para diferentes N elementos

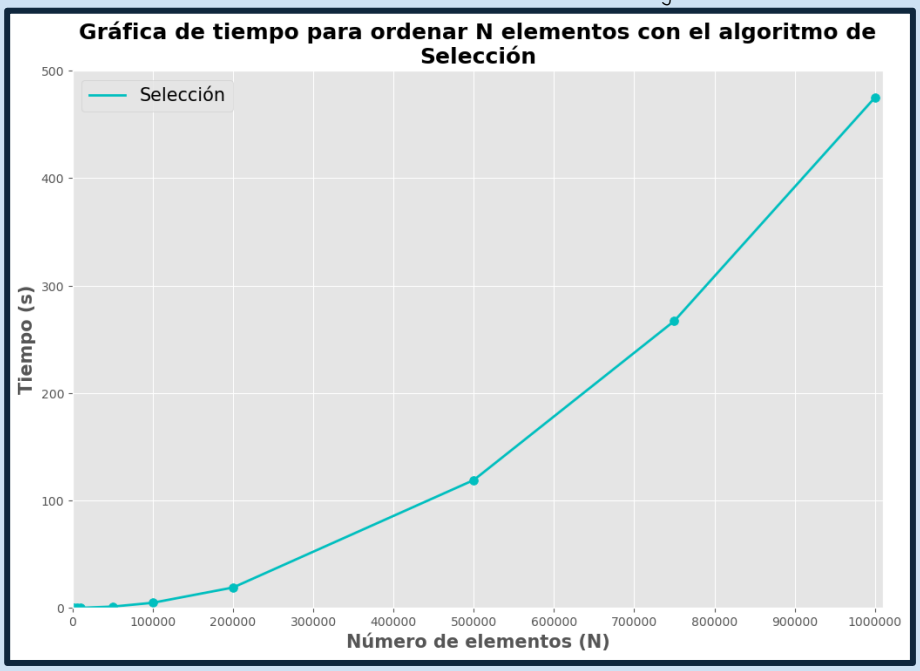
Las marcas de tiempo se realizaron en una computadora con un procesador Ryzen 5 5600X, de 6 núcleos y 12 hilos, con una velocidad de reloj base de 3.7GHz.

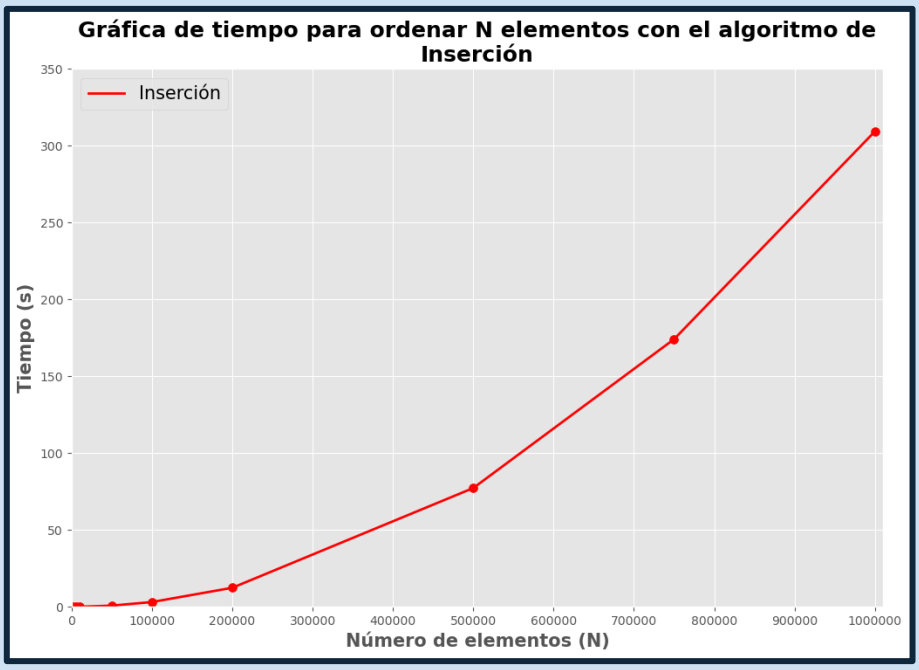
Tiempo que tarda en ordenarse un arreglo (s)						
N elementos	Algoritmo Burbuja	Algoritmo Inserción	Algoritmo Selección	Algoritmo Shell	Algoritmo Mezcla	Algoritmo Rápido
500	0.000311	0.000083	0.000192	0.000048	0.000036	0.000024
1,000	0.000935	0.000314	0.000511	0.000115	0.000096	0.000052
5,000	0.022685	0.007727	0.012051	0.000975	0.000452	0.000301
10,000	0.089389	0.030683	0.047692	0.002410	0.000943	0.000655
50,000	3.318724	0.77485	1.191927	0.018064	0.005370	0.003683
100,000	15.137934	3.106277	4.760601	0.046845	0.011646	0.007811
200,000	62.931618	12.382705	18.969889	0.104611	0.023956	0.015905
500,000	401.362446	77.268103	118.989503	0.297277	0.064052	0.042429
750,000	905.740784	174.118431	267.098923	0.456294	0.097657	0.064370
1,000,000	1608.670611	309.276467	475.025419	0.693272	0.132403	0.087750

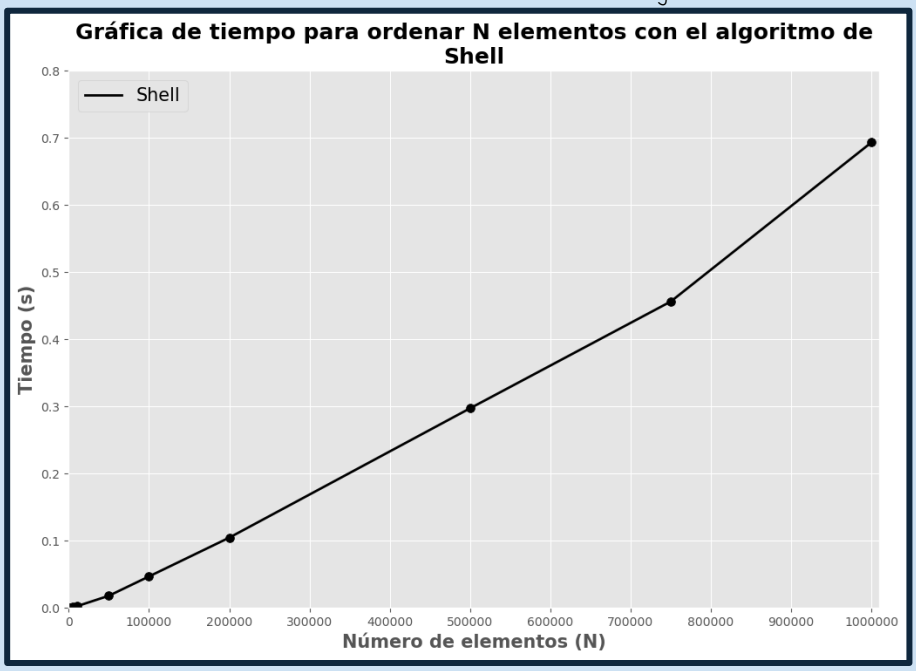
Gráficas de tiempo de ejecución de cada algoritmo

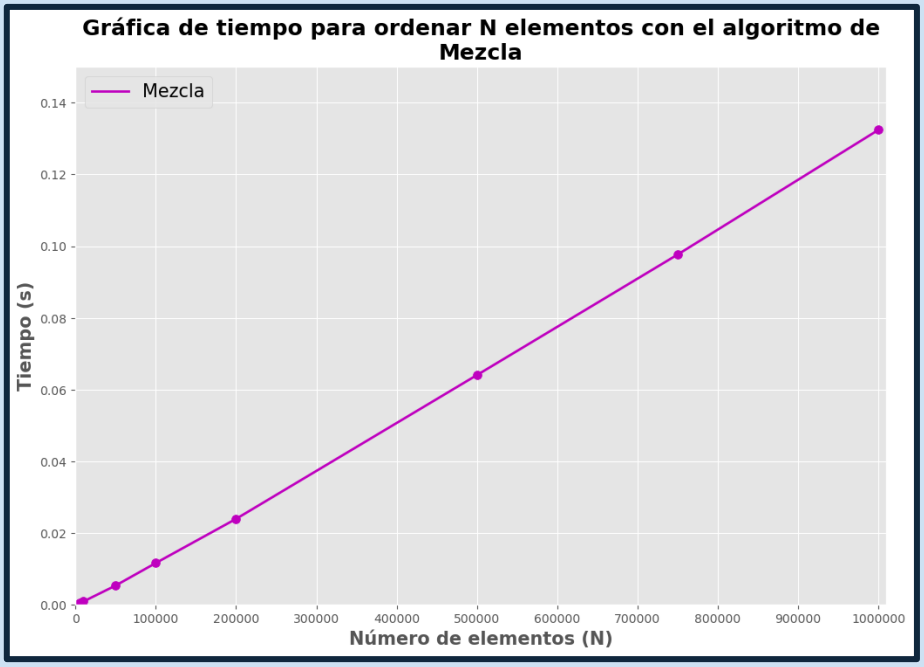
Las gráficas se realizaron en Python con ayuda de la librería Matplotlib.

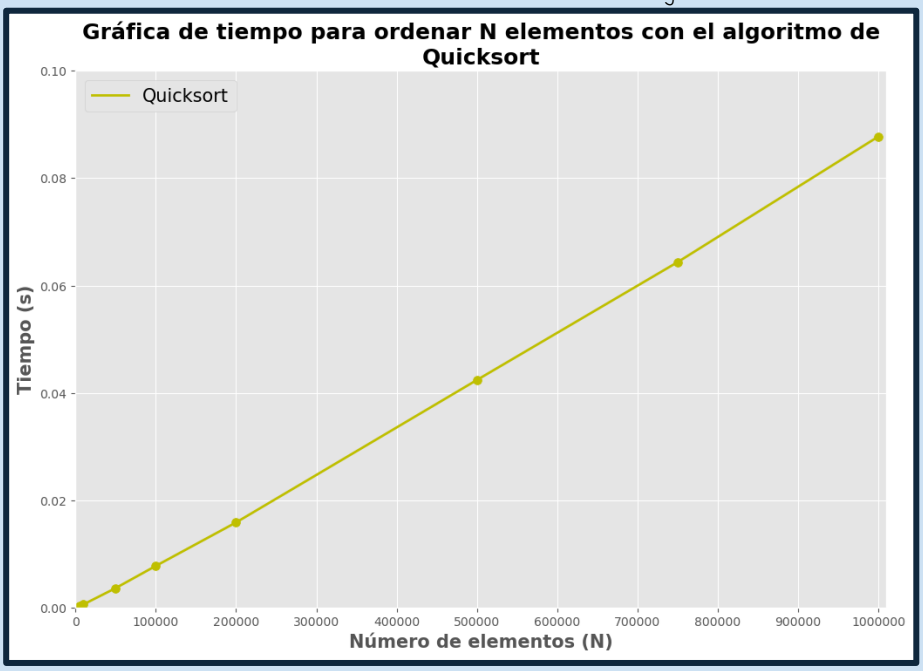



Gráfica de tiempo para ordenar N elementos con el algoritmo de Selección
Selección
Tiempo (s)
Número de elementos (N)
0
100
200
300
400
500
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000


Gráfica de tiempo para ordenar N elementos con el algoritmo de Inserción
Inserción
Tiempo (s)
Número de elementos (N)
0
50
100
150
200
250
300
350
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000

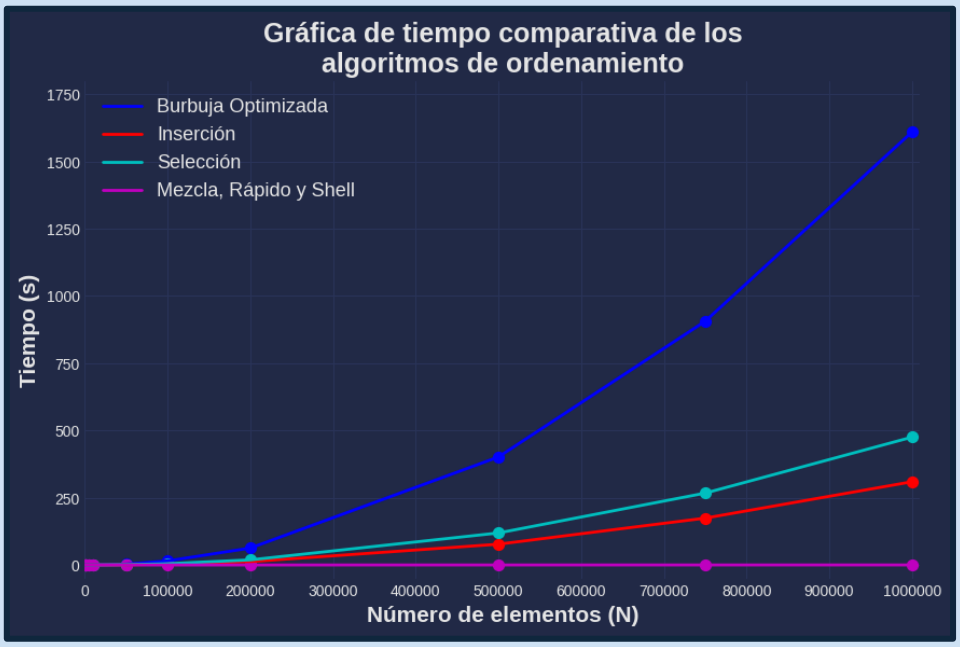

Gráfica de tiempo para ordenar N elementos con el algoritmo de Shell
Shell
Tiempo (s)
Número de elementos (N)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000

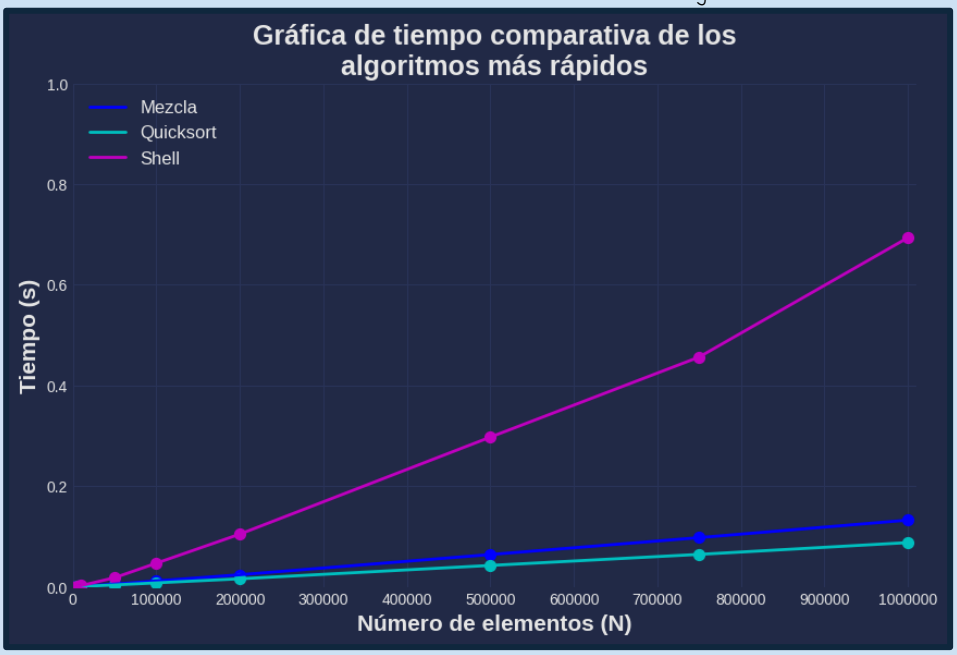

Gráfica de tiempo para ordenar N elementos con el algoritmo de Mezcla
Mezcla
Tiempo (s)
Número de elementos (N)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000



Gráficas comparativas del tiempo de ejecución de cada algoritmo de ordenamiento

Las gráficas se realizaron en Python con ayuda de la librería Matplotlib y un tema personalizado.





¿Cuál fue el algoritmo más eficiente y por qué?

Podemos observar que los algoritmos más rápidos y por mucho son quicksort y mergesort, quicksort teniendo una pequeña diferencia de tiempo que lo hace más rápido, hablando en términos de velocidad, los dos algoritmos son bastante. rápidos, capaces de ordenar 1 millón de elementos en menos de un segundo. Sin embargo, ya que mergesort requiere espacio o memoria extra para los subarreglos, el algoritmo más eficiente es quicksort. Ya que hablando en términos de velocidad y memoria es el mejor, pues las operaciones o intercambios se realizan in situ, es decir: sobre el mismo arreglo.

¿Cuál fue el algoritmo más ineficiente y por qué?

Claramente podemos observar que el algoritmo más ineficiente y por mucho, fue burbuja, que, aunque sea la variante mejorada de burbuja optimizada, no aporta un incremento significativo en su velocidad, llegando a tardar hasta 1600 segundos en ordenar un arreglo con un millón de elementos, tardando 16000 veces más que el algoritmo de ordenamiento por mezcla y rápido.

Dificultades que se tuvieron a la hora de traducir pseudocódigo a C de los algoritmos

Consideramos que no hubo dificultad a la hora de traducir los pseudocódigos a código en C, pues los pseudocódigos fueron bastante claros y explicativos. Aunque hubo una pequeña modificación en el algoritmo de mezcla, ya que el pseudocódigo propuesto en clase no contemplaba el arreglo auxiliar C para almacenar, ordenar y mezclar los subarreglos con el arreglo original.

Errores detectados

Se detectó un error con la implementación del algoritmo de ordenamiento rápido (Quicksort) traducido del pseudocódigo, ya que en casos muy raros y específicos no ordenaba

correctamente los elementos. Más específicamente, en la función pivote, se cambió la condición de los bucles mientras $i < j$ e $i < r$, por $i \leq j$ e $i \leq r$. Con esto se logró solucionar el problema, el problema anterior ocurría porque como ya mencionamos, en casos muy raros donde el pivote fuera el último elemento del extremo derecho de la partición actual, no se consideraba en las operaciones de Quicksort y ocasionaba que las particiones no estuvieran del todo correctamente ordenadas. Nuevamente, como sucedía en casos muy específicos, por ejemplo: al ordenar un millón de elementos, las veces que este caso se presentaba se podía contar con las manos, nuestro equipo se dio cuenta de este detalle comparando la salida que producían los algoritmos y con ayuda de Python comprobar si era la misma para ambos, analizando detenidamente la diferencia de salida se llegó a esta conclusión.

Posibles mejoras

Consideramos que se realizó una buena reducción de código al incluir en una sola función principal (archivo “main.c”) toda la lógica para llamar a cualquier algoritmo de ordenamiento, empleando punteros de funciones y sentencias if-else. También creemos que los pseudocódigos proporcionados de los algoritmos de ordenamiento para su traducción a código en lenguaje C se encontraban bastante sencillos, con poco código y sin relleno o código innecesario.

Anexo

Instrucciones de compilación:

```
gcc main.c burbuja.c insercion.c seleccion.c mergeSort.c quickSort.c shell.c
-o main -std=c17
```

Implementación de header.h

Este archivo de encabezado sirve como librería para los algoritmos utilizados en el programa, en él se encuentran la declaración de constantes y prototipos de funciones requeridas para el funcionamiento del programa.

```c
#ifndef ORDENAMIENTOS_H
#define ORDENAMIENTOS_H
/* Librería para los archivos o algoritmos utilizados en el programa */

/***Declaración de constantes***/
#define TRUE 1
#define FALSE 0
/*** Prototipos de funciones ***/
void BurbujaOptimizada(int *A, int n);
void Seleccion(int *A, int n);
void Insercion(int *A, int n);
void MergeSort(int *A, int p, int r);
void Merge(int *A, int p, int q, int r);
```

```
void Quicksort(int *A, int p, int r);
int Pivot(int *A, int p, int r);
void Intercambiar(int *A, int i, int j);
void Shell(int *A, int n);

#endif
```

Pseudocódigo de ordenamiento por selección

Implementación de main.c

En este archivo se encuentra toda la lógica del programa, el usuario le proporcionará al programa los argumentos requeridos, como el tamaño de los datos que se leerán, el algoritmo que se quiere emplear para ordenar los elementos y si se desear imprimir el contenido del arreglo o imprimir el tiempo que tardó en ordenarse el arreglo.

```
// Librerías que utiliza el programa
#include <stdio.h>   // Librería de entrada y salida estándar
#include <stdlib.h>  // Librería de utilidades estándar
#include <time.h>    // Librería de tiempo
#include "header.h"  // Librería con variables y definiciones de funciones

int main(int n_arg, char *args[]) {
    // Declaración de variables
    int n, f, *A; char opcion, *nombreAlgoritmo;
    // Apuntadores de funciones que apuntan a NULL
    void (*algoritmoFuncion1)(int*, int)=NULL, (*algoritmoFuncion2)(int*, int, int)=NULL;

   // Si el número de argumentos proporcionado al programa es diferente de lo requerido, se sale del programa
    if(n_arg != 4) {
       printf("Indique el tamanio de n, la inicial del algoritmo de ordenamiento a utilizar y si");
       printf("desea imprimir el contenido del arreglo ordenado - Ejemplo: [user@equipo]$ %s 1000 b 0\n", args[0]);
       exit(1);
    }

    // Asignación de variables
    n=atoi(args[1]);
    f=atoi(args[3]);
    opcion=args[2][0];
    A=malloc(n*sizeof(int));

    // Si no se pudo reservar memoria dinámica...
    if(A==NULL){
        printf("No se pudo reservar memoria dinámica para el arreglo\n");
        exit(1);
    }
```

```
    // El puntero de funciones apunta a la función de ordenamiento requerida por el usuario
    if(opcion == 'b') {
        nombreAlgoritmo="de 'Burbuja Optimizada'";
        algoritmoFuncion1 = BurbujaOptimizada;
    }
    else if(opcion == 's') {
        nombreAlgoritmo="por 'Seleccion'";
        algoritmoFuncion1 = Seleccion;
    }
    else if(opcion == 'i') {
        nombreAlgoritmo="por 'Insercion'";
        algoritmoFuncion1 = Insercion;
    }
    else if(opcion == 'm') {
        nombreAlgoritmo="por 'Mezcla'";
        algoritmoFuncion2 = MergeSort;
    }
    else if(opcion == 'q') {
        nombreAlgoritmo="'Rapido'";
        algoritmoFuncion2 = Quicksort;
    }
    else if(opcion == 'h'){
        nombreAlgoritmo="Shell";
        algoritmoFuncion1 = Shell;
    }
    // Si los argumentos introducidos fueron diferentes a los requeridos por el programa
    else {
       printf("Indique el tamanio de n y la inicial del algoritmo de ordenamiento a utilizar - Ejemplo: [user@equipo]$ %s 1000 b 0\n", args[0]);
       exit(1);
    }

    // Se leen los datos necesarios para el programa
    for(int i=0; i<n; i++){
        scanf("%d", &A[i]);
    }

    /* MEDICIÓN DEL TIEMPO */
    // Declaración de variables para la medición de tiempo
    clock_t t_inicio, t_final;
    double t_intervalo;

    // Se llama al apuntador de función
    if(algoritmoFuncion1 != NULL) {
```

```
        // Inicia medición del tiempo
        t_inicio = clock();
        // Se llama al algoritmo de ordenamiento
        (*algoritmoFuncion1)(A, n);
    }
    else {
        // Inicia medición del tiempo
        t_inicio = clock();
        // Se llama al algoritmo de ordenamiento
        (*algoritmoFuncion2)(A, 0, n-1);
    }

    // Termina medición del tiempo
    t_final = clock();
    // Si se desea imprimir el tiempo que tarda en ejecutarse el algoritmo
    if(!f){
        //Cálculo del tiempo y envío de mensaje a la salida estándar con la medición
        t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
        printf("Ordenamiento %s para '%d' elementos. ", nombreAlgoritmo, n);
        printf("Tiempo medido: '%.10f' segundos.\n", t_intervalo);
    }

    // Si se desea imprimir el arreglo
    if(f) {
        for(int i=0; i<n; i++) {
            printf("A[%d]: %d\n", i, A[i]);
        }
    }
    free(A);   // Se libera la memoria reservada para el arreglo dinámico
    return 0;
}
```

Algoritmo de ordenamiento rápido

Ejecución del programa

El programa funciona tomando los siguientes argumentos de ejecución

1. Tamaño de los datos a leer (N)
2. Inicial del algoritmo de ordenamiento a emplear (b, i, s, h, m, q)
3. Imprimir contenido del arreglo o el tiempo que tardó la ejecución del algoritmo (1 o 0, respectivamente).

A continuación se muestra un ejemplo de la ejecución imprimiendo los últimos 10 números de N datos de los arreglos ordenados y el tiempo que tarda en ejecutarse el algoritmo.

Compilación del programa

```
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ gcc main.c burbuja.c insercion.c seleccion.c mergeSort.c quickSort.c shell.c -o main -std=c17
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$
```

Imprimiendo el contenido del arreglo

```
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 100000 b 1 < numeros1M.txt
A[99990]: 2147287994
A[99991]: 2147352342
A[99992]: 2147373031
A[99993]: 2147395653
A[99994]: 2147417281
A[99995]: 2147433626
A[99996]: 2147443379
A[99997]: 2147445108
A[99998]: 2147458290
A[99999]: 2147465711
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 100000 s 1 < numeros1M.txt
A[99990]: 2147287994
A[99991]: 2147352342
A[99992]: 2147373031
A[99993]: 2147395653
A[99994]: 2147417281
A[99995]: 2147433626
A[99996]: 2147443379
A[99997]: 2147445108
A[99998]: 2147458290
A[99999]: 2147465711
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 100000 i 1 < numeros1M.txt
A[99990]: 2147287994
A[99991]: 2147352342
A[99992]: 2147373031
A[99993]: 2147395653
A[99994]: 2147417281
A[99995]: 2147433626
A[99996]: 2147443379
A[99997]: 2147445108
A[99998]: 2147458290
A[99999]: 2147465711
```

```
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 1000000 h 1 < numeros1M.txt
A[999990]: 2147450467
A[999991]: 2147450523
A[999992]: 2147450606
A[999993]: 2147451417
A[999994]: 2147453629
A[999995]: 2147454139
A[999996]: 2147458290
A[999997]: 2147459301
A[999998]: 2147464393
A[999999]: 2147465711
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 1000000 m 1 < numeros1M.txt
A[999990]: 2147450467
A[999991]: 2147450523
A[999992]: 2147450606
A[999993]: 2147451417
A[999994]: 2147453629
A[999995]: 2147454139
A[999996]: 2147458290
A[999997]: 2147459301
A[999998]: 2147464393
A[999999]: 2147465711
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 1000000 q 1 < numeros1M.txt
A[999990]: 2147450467
A[999991]: 2147450523
A[999992]: 2147450606
A[999993]: 2147451417
A[999994]: 2147453629
A[999995]: 2147454139
A[999996]: 2147458290
A[999997]: 2147459301
A[999998]: 2147464393
A[999999]: 2147465711
```

Imprimiendo el tiempo que tardó cada algoritmo

```
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 100000 b 0 < numeros1M.txt
Ordenamiento de 'Burbuja Optimizada' para '100000' elementos. Tiempo medido: '15.2847560000' segundos.
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 100000 i 0 < numeros1M.txt
Ordenamiento por 'Insercion' para '100000' elementos. Tiempo medido: '3.1530730000' segundos.
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 100000 s 0 < numeros1M.txt
Ordenamiento por 'Seleccion' para '100000' elementos. Tiempo medido: '4.3952600000' segundos.
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 1000000 h 0 < numeros1M.txt
Ordenamiento Shell para '1000000' elementos. Tiempo medido: '0.7209530000' segundos.
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 1000000 m 0 < numeros1M.txt
Ordenamiento por 'Mezcla' para '1000000' elementos. Tiempo medido: '0.1344160000' segundos.
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$ ./main 1000000 q 0 < numeros1M.txt
Ordenamiento 'Rapido' para '1000000' elementos. Tiempo medido: '0.0895200000' segundos.
nomad@Android:~/Documents/C_projects/ordenamientoPractica/ordenamientoPractica$
```

Conclusiones

González Joshua: El ordenamiento es un proceso fundamental en la programación y es esencial para muchas aplicaciones, los algoritmos de ordenamiento son una herramienta esencial en la programación para clasificar una gran cantidad de datos y poder manipularlos o procesarlos de una forma más rápida, pues una colección que no se encuentre ordenada siempre será más difícil de tratar o manipular. Sin embargo, cada algoritmo tiene sus propias características y es importante conocer sus ventajas, desventajas, eficiencia y rendimiento en diferentes tamaños de elementos o estado de la colección, por ejemplo: si el número de elementos que necesitamos ordenar es muy grande, si se encuentra completamente en desorden o ya están casi ordenados, también si disponemos de una memoria limitada, etc. En resumen, es imprescindible saber cuándo usar un algoritmo u otro, aunque claro, existen algunos algoritmos que son demasiado ineficientes como para contemplarlos, como el de burbuja optimizada, no se me ocurre mejor ejemplo que lo que ocurre con la función de ordenamiento “sort” de la STL de C++, que se llama introsort, y que en pocas palabras es una combinación del algoritmo de ordenamiento por inserción, heap y Quicksort, así el proceso de ordenamiento será más rápido dependiendo del estado actual del arreglo.

Bibliografía

[1]. S. Makarychev, "Sorting Algorithms Explained with Examples in Python, Java, and C++," freeCodeCamp.org, [En línea]. Disponible en: https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/. [Último acceso: 02 abril 2023].

[2]. "Sorting Algorithms," GeeksforGeeks, [En línea]. Disponible en: https://www.geeksforgeeks.org/sorting-algorithms/. [Último acceso: 02 abril 2023].

[3]. "Sorting Algorithms in Data Structures," Programiz, [En línea]. Disponible en: https://www.programiz.com/dsa/sorting-algorithm. [Último acceso: 02 abril 2023].

[4]. Ramírez García, "Algoritmos de ordenamiento", CIMAT, 2019. [En línea]. Disponible en: https://www.cimat.mx/~alram/comp_algo/clase15.pdf. [Último acceso: 02 abril 2023].

[5]. G. Cervantes González. Ordenamiento por método Shell, 2020.

[6]. "Algoritmo por inserción", GeeksforGeeks, [En línea]. Disponible en: https://www.geeksforgeeks.org/selection-sort/. [Último acceso: 02 abril 2023].

[7]. C. Linares López and F. Berzal Galiano, Algoritmos de ordenamiento. Pearson Educación, 2005.

[8]. S. J. Roldán Betancur, "Algoritmos de ordenamiento," Revista Científica de Ingeniería, vol. 27, no. 1, pp. 76-84, 2014.

[9]. J. Aguirre Espinoza, "Algoritmos de ordenamiento: comparativa y análisis teórico," Revista Científica de Ingeniería, vol. 32, no. 1, pp. 38-47, 2018.

[10]. F. Berzal Galiano and C. Linares López, "Algoritmos de ordenamiento: implementación y análisis comparativo," Revista Científica de Ingeniería, vol. 19, no. 2, pp. 25-32, 2015.

[11]. R. Salinas García, "Análisis de Algoritmos de Ordenamiento," Revista de Investigación Académica, vol. 12, no. 1, pp. 56-65, 2015.