TRABAJO ESCRITO DEL PROYECTO

PROYECTO #1 - COMPLEJIDAD COMPUTACIONAL EN LOS ALGORITMOS DE ORDENAMIENTO

Profesor: Edgar Tista García

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 5

Integrantes: Giron Escalona Erendira Nayely, López Martínez Diana , Zarate Menes Quetzalli

No. de lista: 12, 20, 35

Semestre: 2024-2

Fecha de entrega: 24 de marzo del 2024

Observaciones:

CALIFICACIÓN:

OBJETIVO

Que el alumno observe la complejidad computacional de los algoritmos de ordenamiento para comparar su eficiencia de ejecución en grandes volúmenes de información

INTRODUCCIÓN

En nuestro proyecto, abordamos el estudio teórico y práctico de los algoritmos de ordenamiento, fundamentales en el campo de la ciencia de la computación. Estos algoritmos son esenciales para la organización y manejo eficiente de datos, una habilidad clave en la era de la información.

InsertionSort es un algoritmo intuitivo que construye la salida ordenada elemento por elemento, insertando cada nuevo elemento en su posición correcta dentro de la parte ya ordenada.

SelectionSort mejora este proceso seleccionando el elemento más pequeño de la parte no ordenada y colocándolo al principio de la lista ordenada.

Heapsort transforma la lista en un montículo binario y luego extrae los elementos de manera ordenada, aprovechando la estructura de árbol para mejorar el tiempo de ejecución.

Bubblesort es un método simple que revisa repetidamente la lista para intercambiar elementos adyacentes que están en el orden incorrecto.

Quicksort divide la lista en dos partes, ordenando cada una de ellas de manera independiente a través de un proceso recursivo basado en un pivote.

Mergesort también utiliza la recursividad, dividiendo la lista en mitades hasta llegar a listas de un solo elemento y luego combinándolas de manera ordenada.

Finalmente, exploraremos un algoritmo adicional, que seleccionaremos por su relevancia y eficacia comparativa, para ampliar nuestra comprensión de estas técnicas de ordenamiento.

Nuestro análisis se centrará en cómo estos algoritmos manejan el incremento en el tamaño de los datos, observando la cantidad de operaciones necesarias para alcanzar la lista ordenada final. Este enfoque nos permitirá no solo entender los principios básicos de funcionamiento de cada algoritmo, sino también su comportamiento y rendimiento en condiciones reales y variadas.

DESARROLLO

Menu Principal

En relación al requisito de manejar diferentes tamaños de arreglos, se ha implementado una función denominada “llenarArregloConAleatorios”. En esta función, se utiliza la clase Random que se encuentra en el paquete java.util.

Consideraciones para Inicializar Arreglos

En el contexto de la inicialización de arreglos, nos encontramos con un desafío importante: ¿cómo manejar diferentes tamaños de arreglos de manera eficiente y efectiva? Durante nuestras discusiones, evaluamos diversas alternativas y llegamos a una solución que equilibra la funcionalidad con las limitaciones de recursos.

1. Ciclo de for para Tamaños de Arreglo:
   - Inicialmente, consideramos la posibilidad de utilizar un ciclo iterativo para recorrer una gama de tamaños de arreglo. Esto nos permitiría mostrar los resultados en pantalla y evaluar la complejidad del algoritmo para cada tamaño.
   - Sin embargo, surgió una preocupación importante: ¿qué sucede si la computadora del usuario no puede manejar un arreglo excesivamente grande? La ejecución podría volverse ineficiente o incluso agotar los recursos disponibles.
2. Solución: Menú de Elección de Tamaño:
   - Para abordar esta preocupación, optamos por implementar un menú interactivo. En lugar de asumir un tamaño de arreglo predeterminado, le damos al usuario la capacidad de elegir el tamaño deseado.
   - El menú permite al usuario especificar el número de elementos en el arreglo. Esto garantiza que la prueba de complejidad se realice dentro de los límites de capacidad de su máquina.
3. Función con Parámetro de Tamaño:
   - Para llevar a cabo esta implementación, creamos una función llamada "llenarArregloConAleatorios". Dentro de este menú, se le pasa como parámetro el tamaño de arreglo deseado.

- La función utiliza la clase Random del paquete java.util para generar valores aleatorios y llenar el arreglo con datos representativos.

Consideraciones para Comparar Complejidades de Algoritmos de Ordenamiento.

En nuestro análisis riguroso de los algoritmos de ordenamiento, nos encontramos con un desafío fundamental: ¿cómo comparar de manera justa las complejidades de diferentes métodos en un escenario realista? Para abordar esta cuestión, diseñamos una estrategia que garantiza una evaluación imparcial y significativa.

1. El Dilema del Mejor Caso:
   - Al comparar algoritmos, es crucial considerar el mejor caso, el peor caso y el caso promedio.
   - Siempre que enviáramos un arreglo ya ordenado a un nuevo método de ordenamiento, estaríamos en el mejor caso. Sin embargo, esto no reflejaría la realidad, ya que en situaciones reales, rara vez se reciben arreglos perfectamente ordenados.
2. Solución: Arreglos Desordenados y Copias:
   - Para abordar este sesgo, creamos dos copias del mismo arreglo:
     - Arreglo A (Original): Generado aleatoriamente con valores desordenados.
     - Arreglo B (Copia): Inicialmente idéntico a A, pero se le asignará el resultado de un algoritmo de ordenamiento.
   - Ejecutamos cada algoritmo de ordenamiento en B, lo que

garantiza que estemos evaluando el caso promedio.

3. Repetibilidad y Tamaño del Arreglo:
   - El usuario selecciona el tamaño del arreglo en el menú interactivo.
   - Al repetir el proceso con el mismo arreglo desordenado, aseguramos la consistencia y la comparabilidad.

En resumen, nuestra metodología garantiza un análisis imparcial y realista de los algoritmos de ordenamiento. Al utilizar arreglos desordenados y copias, evitamos el sesgo del mejor caso y obtenemos una visión más precisa de la eficiencia de cada método.

InsertionSort

En la optimización del algoritmo *InsertionSort*, se introdujeron contadores para monitorear las operaciones de inserción, intercambio y comparación. Estos contadores son cruciales para analizar la complejidad del algoritmo, ya que proporcionan una cuantificación directa de las operaciones elementales que realiza.

Para integrar estos contadores, se realizaron las siguientes modificaciones en el código:

- En la estructura `while`, que evalúa si el índice `j` es mayor o igual a cero y si el elemento en la posición `j` es mayor que el valor en el índice `index`, se incrementa el contador de comparaciones. Esto se debe a que en cada iteración de esta estructura se realiza una comparación.
- Si la condición de comparación es verdadera, se procede a intercambiar los valores del arreglo en las posiciones `j+1` y `j`. En este punto, se incrementa el contador de intercambios, reflejando la

realización de esta operación.

- La última modificación consistió en añadir el contador de inserciones, el cual se incrementa después de la estructura `while`. Esto indica que se ha encontrado la posición final para el valor que se está insertando, y por lo tanto, se contabiliza una inserción.

La implementación de estos contadores no presentó dificultades significativas, debido a la simplicidad inherente al algoritmo de *InsertionSort*. La adición de estas métricas permite un análisis más detallado y preciso de la complejidad del algoritmo, facilitando la comprensión de su comportamiento y eficiencia en diferentes escenarios de ordenamiento.

SelectionSort

Algoritmo de Selection Sort

El algoritmo de **Selection Sort** sigue un enfoque simple y directo para ordenar una lista de elementos. Funciona de la siguiente manera:

1. Encuentra el elemento más pequeño en la lista.
2. Intercambia el elemento más pequeño con el primer elemento.
3. Busca el siguiente elemento más pequeño y lo intercambia con el segundo elemento.
4. Repite el proceso hasta que todos los elementos estén ordenados.

Este algoritmo divide el conjunto de elementos en dos partes: la parte ordenada (izquierda) y la parte no ordenada (derecha). En cada iteración, busca el elemento más pequeño en la parte no ordenada y

lo coloca en su posición correcta en la parte ordenada. A pesar de su simplicidad, **Selection Sort** puede ser ineficiente para grandes conjuntos de datos debido a su complejidad cuadrática.

Funcionalidad de la Clase Selection Sort

La clase **Selection Sort** contiene el método `selection`, que toma un arreglo de enteros como entrada y representa la implementación del algoritmo. La funcionalidad se describe a continuación:

- Inicializa tres variables para llevar un registro del número de iteraciones, comparaciones e intercambios realizados durante el proceso de ordenamiento.
- Obtiene la longitud del arreglo para determinar el número de elementos a ordenar.
- Establece un bucle exterior que se ejecuta para cada elemento del arreglo, excepto el último.
- Dentro del bucle, incrementa el contador de iteraciones.
- Inicializa la variable `min` con el índice actual del bucle exterior.
- Ejecuta un bucle interior para cada elemento restante en el arreglo.
- Incrementa el contador de comparaciones cada vez que se compara un elemento.
- Si encuentra un elemento menor, actualiza la posición del mínimo.
- Intercambia el mínimo elemento encontrado con el primer elemento de la parte no ordenada.

- Incrementa el contador de intercambios al realizar un intercambio de elementos.
- Cierra los ciclos y presenta los resultados utilizando la función `printArray`.

Las modificaciones realizadas al algoritmo de **Selection Sort** han agregado contadores para registrar el número de *iteraciones*, *comparaciones* e *intercambios* realizados durante el proceso de ordenamiento. Estos contadores se han incorporado en bucles adecuados dentro del algoritmo. La variable *itera* se incrementa en cada iteración del bucle externo, *comp* se incrementa cada vez que se realiza una comparación entre elementos, y *swaps* se incrementa cada vez que se realiza un intercambio de elementos. Estos contadores proporcionan una forma de medir el rendimiento del algoritmo y entender mejor su comportamiento en diferentes conjuntos de datos.

HeapSort

En el desarrollo del algoritmo de ordenamiento por montículo (*HeapSort*), se presentaron desafíos significativos para contabilizar las operaciones de comparación e intercambio. La complejidad radica en la naturaleza recursiva y la manipulación del arreglo durante la construcción y el ordenamiento del montículo.

Para abordar estas dificultades, se introdujeron dos contadores estáticos, `INTERCAMBIOS` y `COMPARACIONES`, que incrementan su valor en puntos estratégicos del algoritmo. En la función `heapifyASC`, se incrementa `COMPARACIONES` cada vez que se evalúa una condición para determinar el nodo más grande. Asimismo, `INTERCAMBIOS` se incrementa cada vez que se realiza un intercambio de elementos en el

arreglo.

La contabilización de estas operaciones se realiza de esta forma debido a que proporciona una medida directa y clara del trabajo realizado por el algoritmo. Las comparaciones son indicativas de la cantidad de decisiones tomadas, mientras que los intercambios reflejan las modificaciones efectuadas sobre la estructura del montículo. Juntos, estos contadores ofrecen una visión integral del esfuerzo computacional requerido para ordenar el arreglo.

El método `imprimirContadores` se encarga de mostrar la suma de ambos contadores, ofreciendo así el número total de operaciones realizadas. Posteriormente, se reinician los contadores para evitar la acumulación de operaciones entre ejecuciones sucesivas del algoritmo.

En conclusión, el seguimiento de estas operaciones es fundamental para evaluar el rendimiento del algoritmo de *HeapSort*, y los contadores implementados facilitan esta tarea de manera eficiente y sistemática.

BubbleSort

El algoritmo de **Bubble Sort** cuenta con un enfoque simple y directo para ordenar una lista de elementos. Comienza comparando cada par de elementos adyacentes en la lista y los intercambia si están en el orden incorrecto. De esta manera, el elemento más grande "burbujea"hacia la parte superior de la lista en cada iteración. Este proceso se repite hasta que la lista esté completamente ordenada y ningún intercambio adicional sea necesario. Aunque es fácil de implementar y entender, **Bubble Sort** puede ser ineficiente para grandes conjuntos de datos debido a su complejidad cuadrática.

En cuanto a la funcionalidad, la clase **Bubble Sort** contiene lo siguiente:

- El método `bubbleSort`, que toma un arreglo de enteros como parámetro y lo ordena utilizando el algoritmo **Bubble Sort**.
  - Primero se obtiene la longitud del arreglo, lo que permite saber cuántos elementos hay que ordenar.
  - Se inicializan tres variables para llevar un registro del número de iteraciones (*iteraciones*), comparaciones (*comparaciones*) e intercambios (*intercambios*) realizados durante el proceso de ordenamiento.
  - Se inicia un bucle externo que se encarga de iterar sobre cada elemento del arreglo, comenzando desde el último y avanzando hacia el primero.
  - Dentro del bucle externo, se crea un segundo bucle para iterar sobre cada par de elementos adyacentes en el arreglo.
  - Se establece una condición dentro del segundo bucle que verifica si el elemento actual es mayor que el siguiente en el arreglo.
  - Si el elemento actual es mayor que el siguiente, se llama a la función `swap` para intercambiar los elementos y ordenarlos.
  - Se cierran los ciclos y se presentan los resultados, auxiliándose de `printArray` para mostrar la lista antes y después de ordenar.

Las modificaciones para el conteo de comparaciones en el algoritmo de **Bubble Sort** proporcionan una manera de evaluar y analizar su rendimiento. Al agregar contadores para estas operaciones, se puede monitorear cuántas veces se ejecuta cada paso crucial del algoritmo

durante su ejecución. Esto permite cuantificar el rendimiento del algoritmo y compararlo con otros algoritmos de ordenamiento. Además, estas métricas ayudan a comprender mejor el comportamiento del algoritmo en diferentes conjuntos de datos y permiten identificar áreas para posibles mejoras o optimizaciones.

QuickSort

La modificación principal fue agregar a los contadores las operaciones de comparaciones, intercambios e inserciones. Estas sirven para analizar la complejidad del algoritmo y compararlos con los otros algoritmos.

Para modificar el algoritmo y agregar estos contadores se realizó lo siguiente:

- Dentro del método `partition` se agregan los contadores de las operaciones, para los intercambios, comparaciones e inserciones, los intercambios se implementan dentro del ciclo `for` después de haber verificado que el elemento en la posición j del arreglo es decir el actual sea menor o igual al pivote se intercambian los elementos que se encuentran en las posiciones de los índices i y j, se usó también al método `swap` el cual se encuentra en las utilerías y después de ello se usan los contadores de intercambios y comparaciones pues se están realizando ambas operaciones en este momento. Por último se agregó al contador de inserciones el cual se usó después de realizar la inserción del elemento usando el método `swap` el cual se encuentra en las utilerías. Ya que se debe incrementar cada que se realice esta operación.

La dificultad que se presentó fue que el número de los contadores

se almacenaban conforme se utilizaba el algoritmo, para intentar solucionar esto creé un método para que reiniciara a los contadores y no se acumularan las operaciones. Sin embargo, esta no era la mejor solución, mi compañera Quetzalli me comentó que si se inicializan los contadores desde el principio en 0 no sería necesario reiniciarlos y desaparece el problema de acumulación de operaciones.

MergeSort

Reflexiones sobre el Conteo de Operaciones en MergeSort

Durante mi análisis exhaustivo de los algoritmos de ordenamiento, me encontré con un desafío fundamental: ¿cómo comparar de manera justa las complejidades de diferentes métodos en un escenario realista? Permíteme compartir mis reflexiones al respecto:

1. Dificultades Iniciales:
   - Al evaluar la eficiencia de MergeSort, necesitaba una forma objetiva de medir su rendimiento.
   - Contar las operaciones (comparaciones e inserciones) se convirtió en un aspecto crucial para evaluar su eficiencia.
2. Resolución: Conteo de Operaciones
   - Introduje dos variables estáticas: INSERCIONES y COMPARACIONES.
   - Cada vez que se realiza una comparación entre elementos del arreglo, incremento COMPARACIONES.
   - Cada vez que se inserta un elemento en el arreglo auxiliar durante la fusión, incremento INSERCIONES.

3. Razón detrás del Conteo:
    - Comparaciones: Representan la cantidad de veces que evaluamos si un elemento es mayor o menor que otro. Esto ayuda a medir la eficiencia en términos de decisiones tomadas.
    - Inserciones: Representan la cantidad de veces que copiamos un elemento en el arreglo auxiliar durante la fusión. Esto refleja la cantidad de operaciones de escritura realizadas.
4. Uso en la Evaluación:
    - Al imprimir los contadores al final de la ejecución, obtengo una visión completa del número total de operaciones realizadas.
    - Esto me permite comparar MergeSort con otros algoritmos y evaluar su eficiencia en diferentes contextos.

Adicional-CountingSort

Dificultades en el Conteo de Operaciones en Counting Sort

Como desarrollador, enfrenté varias dificultades al contar las operaciones realizadas en el algoritmo de ordenamiento por conteo (Counting Sort) que se presenta a continuación. A continuación, describo estas dificultades, cómo se resolvieron y por qué se contabilizan las operaciones de esa manera.

1. **Bucles Anidados:**
    - El código contiene varios bucles anidados, lo que dificulta contar con precisión las operaciones.
    - El bucle más externo inicializa el arreglo de conteo, el se-

gundo bucle cuenta la frecuencia de cada dígito y el tercer bucle reemplaza los elementos originales del arreglo según la frecuencia.

- Cada bucle contribuye al recuento total de operaciones, pero hacer un seguimiento de todas las iteraciones anidadas es complejo.

2. **Incremento de Contadores Dentro de Bucles:**
   - Los contadores (INSERCIONES, ITERACIONES y COMPARACIONES) se incrementan dentro de los bucles.
   - Por ejemplo, en la función findMax, los contadores se incrementan dentro del bucle que busca el valor máximo.
   - Esto dificulta separar las operaciones relacionadas con cada tarea específica (por ejemplo, encontrar el valor máximo, contar frecuencias y reemplazar elementos).
3. **Operaciones Mixtas:**
   - El código realiza diversas operaciones (asignaciones, comparaciones e incrementos) dentro de los mismos bucles.
   - Por ejemplo, el bucle while dentro del tercer bucle realiza tanto asignaciones A[i++] = j como decrementos (count[j]–).
   - Contar estas operaciones mixtas con precisión es complicado.
4. **Sentencias Condicionales:**
   - El código incluye sentencias condicionales (por ejemplo, if (A[i] ¿max)) que afectan el recuento de operaciones.
   - Estas condiciones conducen a diferentes caminos en el flujo de ejecución, lo que dificulta rastrear el número exacto de operaciones.

Resolución del Conteo

Para abordar estas dificultades, coloqué los contadores en puntos estratégicos del código:

- **INSERCIONES**: Se incrementa al reemplazar elementos en el arreglo original.
- **ITERACIONES**: Se incrementa en bucles (tanto en bucles de inicialización como de conteo).
- **COMPARACIONES**: Se incrementa durante las comparaciones (por ejemplo, al encontrar el valor máximo).

El recuento final de operaciones se obtiene sumando estos tres contadores. Los métodos imprimirContadores y retornarContadores proporcionan el recuento total de operaciones.

Razón para Contar las Operaciones de Esta Manera

Los contadores elegidos (inserciones, iteraciones y comparaciones) representan operaciones comunes en algoritmos de ordenamiento:

- Contar inserciones ayuda a rastrear cuántas veces se mueven los elementos.
- Contar iteraciones captura la complejidad general de los bucles.
- Contar comparaciones es esencial para comprender la eficiencia del algoritmo.

Al sumar estos contadores, obtenemos una visión completa del rendimiento del algoritmo.

IMPLEMENTACIÓN

En cuanto a HeapSort, MergeSort y CountingSort EJECUCIONES: Para determinar el número de operaciones y el promedio de cada uno.

```
¿Que tamaño de arreglo te gustaría probar?
50

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 598
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 797
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 20063

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 604
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 785
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 18951

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 590
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 790
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 19835

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 612
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 787
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 19838
```

```
                EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 588
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 792
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 19807

Promedio ejecucionesHeapSort: 598       Promedio ejecucionesMergueSort: 790     Promedio ejecucionesCountingSort: 19698
```

```
¿Que tamaño de arreglo te gustaría probar?
100

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 1420
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 1875
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 20212

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 1418
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 1885
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 20195

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 1414
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 1883
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 20217

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 1414
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 1884
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 20219
```

```
                EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 1415
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 1881
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 20096

Promedio ejecucionesHeapSort: 1416      Promedio ejecucionesMergueSort: 1881    Promedio ejecucionesCountingSort: 20187
```

```
¿Que tamaño de arreglo te gustaría probar?
500

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 10042
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 12819
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 21441

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 10062
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 12819
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 21471

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 10065
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 12834
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 21475

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 10014
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 12829
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 21464
```

```
                EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 10056
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 12848
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 21413

Promedio ejecucionesHeapSort: 10047     Promedio ejecucionesMergueSort: 12829   Promedio ejecucionesCountingSort: 21452
```

```
¿Que tamaño de arreglo te gustaría probar?
800

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 17437
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 22261
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22395

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 17409
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 22252
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22399

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 17427
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 22278
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22361

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 17481
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 22274
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22397
```

```
             EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 17424
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 22274
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22405

Promedio ejecucionesHeapSort: 17435     Promedio ejecucionesMergueSort: 22267   Promedio ejecucionesCountingSort: 22391
```

```
¿Que tamaño de arreglo te gustaría probar?
1000

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 22698
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 28656
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22978

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 22513
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 28676
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22991

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 22382
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 28640
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22986

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 22591
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 28674
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 23006
```

```
              EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 22507
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 28633
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 22985

Promedio ejecucionesHeapSort: 22538     Promedio ejecucionesMergueSort: 28655   Promedio ejecucionesCountingSort: 22989
```

```
¿Que tamaño de arreglo te gustaría probar?
2000

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 50135
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 63356
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 26000

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 50235
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 63330
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 25993

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 50183
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 63235
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 26000

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 50369
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 63297
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 25993
```

```
                EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 50134
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 63314
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 26004

Promedio ejecucionesHeapSort: 50211     Promedio ejecucionesMergueSort: 63306   Promedio ejecucionesCountingSort: 25998
```

```
¿Que tamaño de arreglo te gustaría probar?
5000

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 141814
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 178935
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 35002

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 141756
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 178813
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 35008

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 141809
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 178818
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 35007

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 141710
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 178802
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 35003
```

```
               EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 141865
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 178831
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 35004

Promedio ejecucionesHeapSort: 141790    Promedio ejecucionesMergueSort: 178839  Promedio ejecucionesCountingSort: 35004
```

```
¿Que tamaño de arreglo te gustaría probar?
10000

                EJECUCIÓN NO 1
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 308832
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 387721
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 50004

                EJECUCIÓN NO 2
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 308238
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 387692
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 50002

                EJECUCIÓN NO 3
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 308493
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 387665
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 50006

                EJECUCIÓN NO 4
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 308509
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 387635
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 50008
```

```
                EJECUCIÓN NO 5
***ORDENAMIENTO CON Heapsort***
El número de operaciones es: 308670
***ORDENAMIENTO CON MergeSort***
El número de operaciones es: 387618
***ORDENAMIENTO CON CountingSort***
El número de operaciones es: 50005

Promedio ejecucionesHeapSort: 308548    Promedio ejecucionesMergueSort: 387666  Promedio ejecucionesCountingSort: 50005
```

Tabla HeapSort	
Tamaño del arreglo	Cantidad de operaciones
50	598
100	1416
500	10047
800	17435
1000	22538
2000	50211
5000	141790
10000	308548

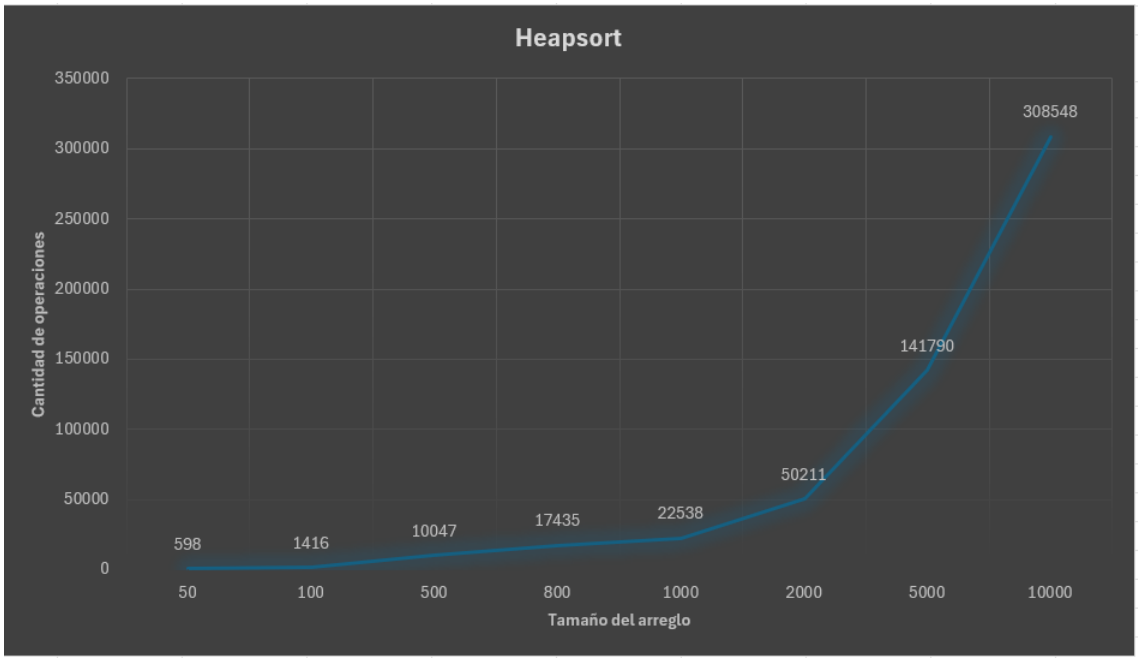


Tabla MergeSort	
Tamaño del arreglo	Cantidad de operaciones
50	790
100	1881
500	12829
800	22267
1000	28655
2000	63306
5000	178839
10000	387666

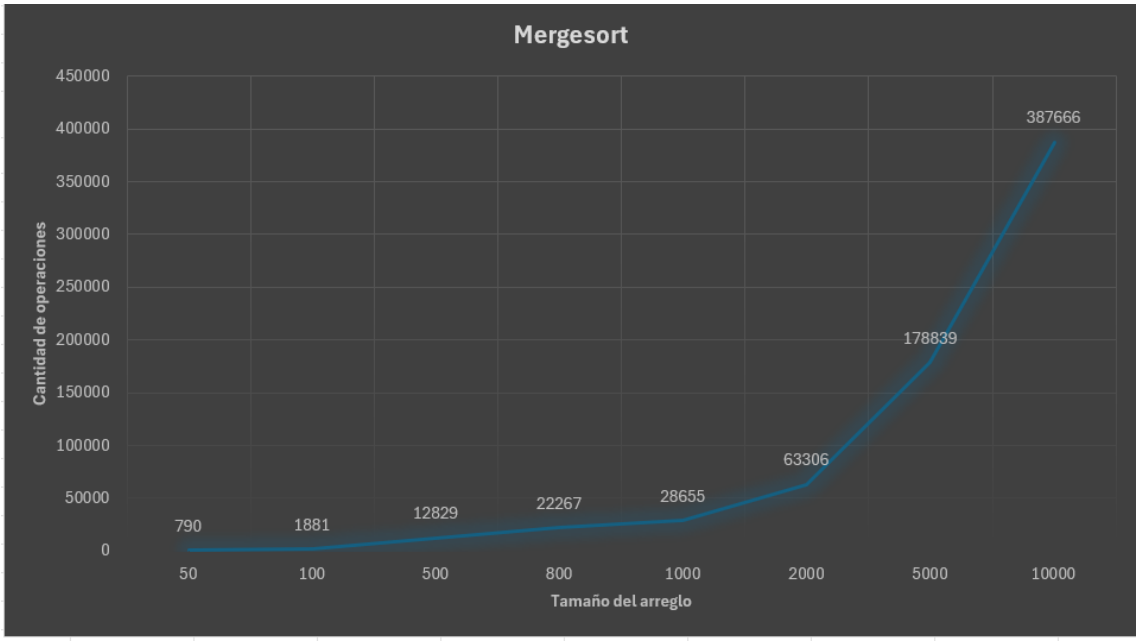
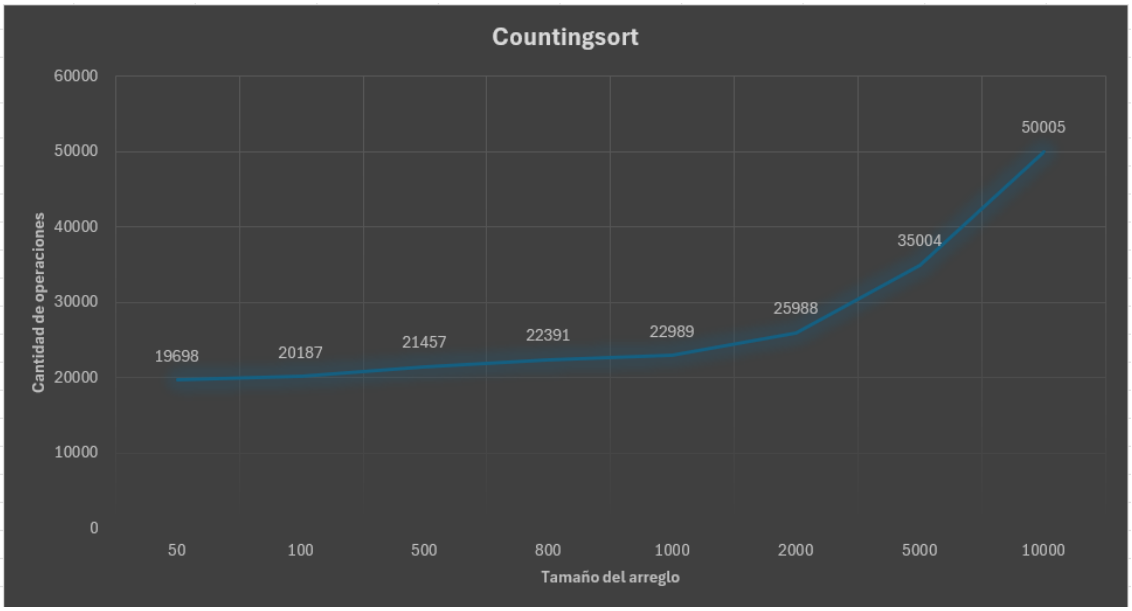


Tabla Countingsort	
Tamaño del arreglo	**Cantidad de operaciones**
50	19698
100	20187
500	21457
800	22391
1000	22989
2000	25988
5000	35004
10000	50005



Análisis Comparativo de Algoritmos de Ordenamiento

En el estudio exhaustivo de los algoritmos de ordenamiento, hemos observado patrones interesantes al comparar repetidamente MergeSort, HeapSort y CountingSort. Estas observaciones nos han proporcionado valiosas perspectivas sobre la eficiencia relativa de estos métodos.

1. Disparidad en el Rendimiento
    - Después de realizar múltiples ejecuciones con diferentes tamaños de arreglos, notamos una disparidad significativa en el número de operaciones realizadas por cada algoritmo.
    - Específicamente, cuando superamos un rango de tamaño de arreglo (aproximadamente 1000 elementos), CountingSort se destaca como más eficiente en términos del número total de operaciones en comparación con los otros dos algoritmos.
2. El Dilema de los Datos Dispersos:
    - Para comprender esta diferencia, consideremos un ejemplo concreto. Supongamos que tenemos una tabla de cinco elementos con datos que varían de 1000 a 9999.
    - El arreglo original tiene solo cinco elementos, mientras que el arreglo de conteo abarca un rango mucho más amplio (de 1000 a 9999).
    - Cuando el tamaño del arreglo a ordenar supera los 1000 elementos, notamos que el arreglo de conteo se ajusta gradualmente al tamaño del arreglo original. Esto se debe a la dispersión de los datos en el rango más amplio.
3. Complejidad de Operaciones y Tiempo:
    - La complejidad de operaciones en CountingSort es similar a la complejidad de tiempo, expresada como O(n + k).
    - Aquí, n representa el tamaño del arreglo original, y k es el rango de valores posibles. A medida que n aumenta, la influencia de k se vuelve más evidente.

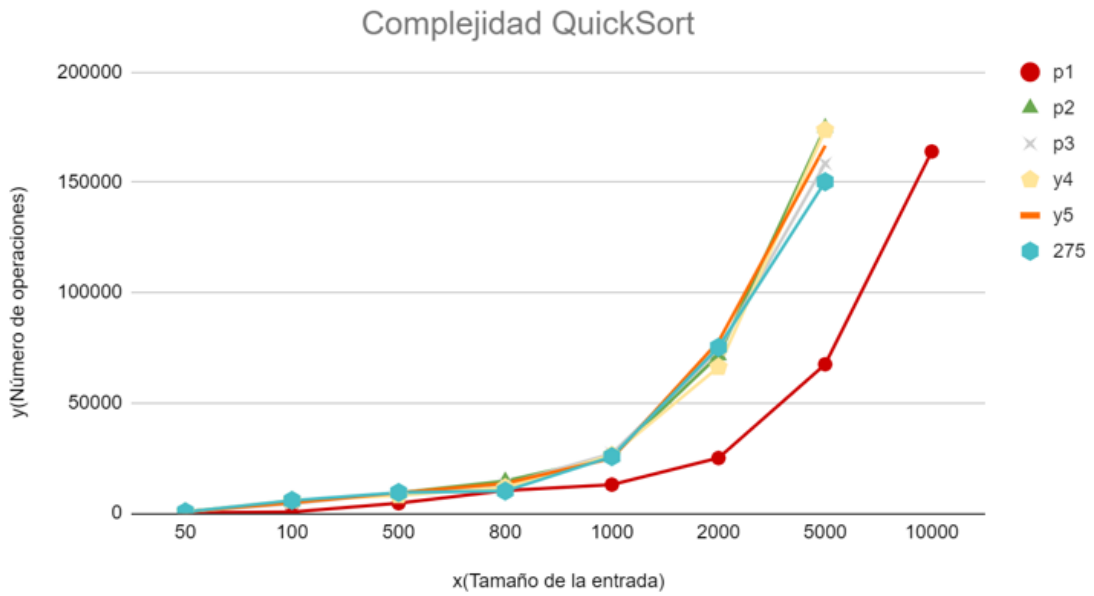
Influencia de la Dispersión de Datos en los Algoritmos de Ordena-

miento

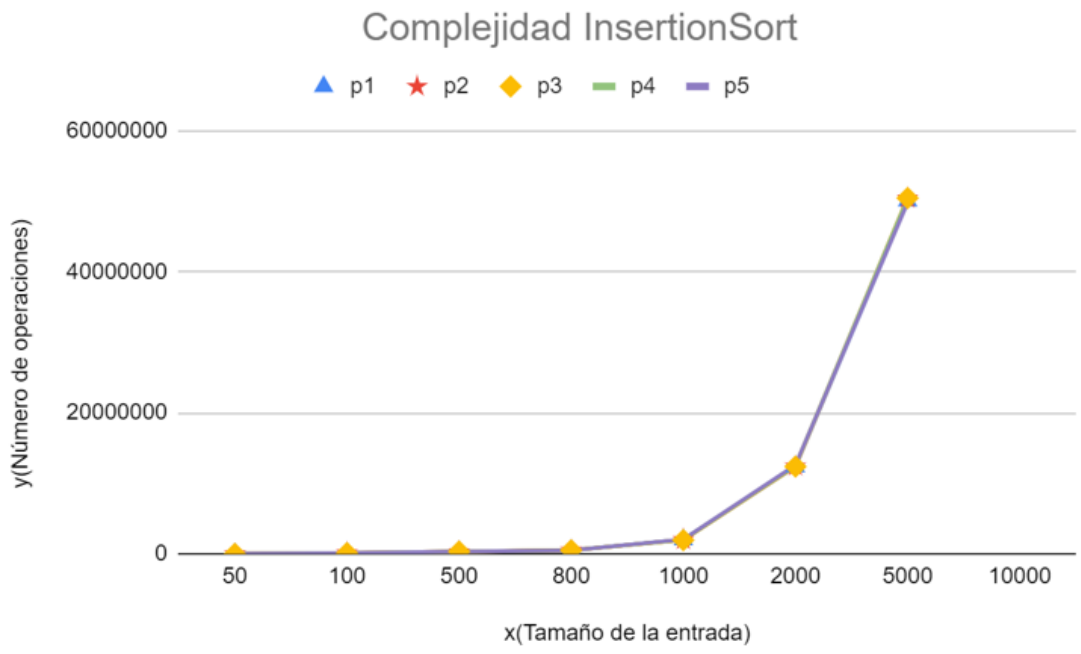
En el análisis profundo de los algoritmos de ordenamiento, surge una reflexión importante: la dispersión de los datos ejerce una influencia significativa en la elección del método adecuado. Consideremos este aspecto con mayor detalle.

1. Rango de Valores y Eficiencia:
   - La variabilidad en los valores presentes en un conjunto de datos afecta directamente la eficiencia de los algoritmos de ordenamiento.
   - En particular, cuando se trata de algoritmos como Counting-Sort, el rango de los números desempeña un papel crucial.
2. CountingSort y su Sensibilidad al Rango:
   - CountingSort es altamente sensible al rango de valores. Su complejidad está directamente relacionada con la cantidad de valores únicos en el conjunto.
   - Si el rango es pequeño (es decir, los valores están concentrados en un intervalo estrecho), CountingSort se vuelve más eficiente.
3. Comparación con Otros Algoritmos:
   - En contraste, algoritmos como MergeSort y HeapSort no son tan sensibles al rango. Su complejidad depende principalmente del tamaño del arreglo (n).
   - Cuando el rango es pequeño y se acerca a n, CountingSort supera a los otros algoritmos en términos de operaciones realizadas.

En cuanto a QuickSort e InsertionSort:

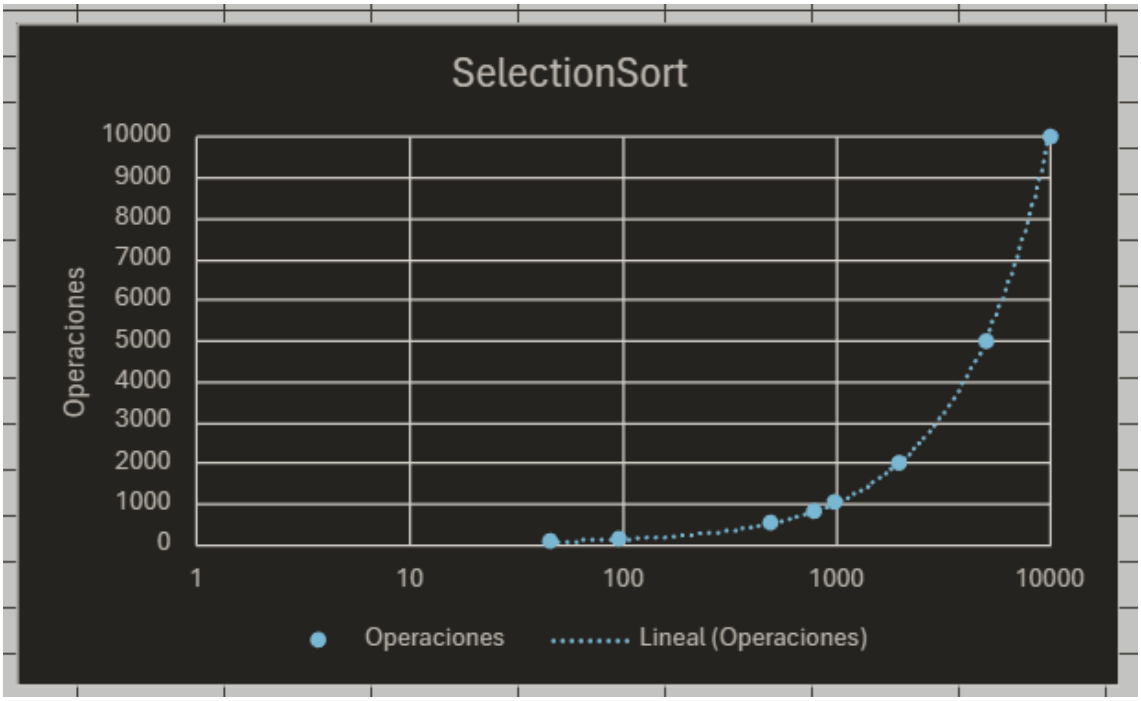


Se realizaron 6 ejecuciones para analizar el algoritmo `quickSort`, el algoritmo `quickSort` tiene una complejidad para el mejor caso de $n \log n$, en la gráfica que se muestra se observa que la complejidad del algoritmo se tuvo para el mejor caso. Esto también se verificó manualmente; un ejemplo es que para la línea de color rojo que se muestra en la gráfica, la cantidad de operaciones realizadas para el arreglo de tamaño: 50, 100, 500, 800, 1000, 2000, 5000 y 10000, fueron los siguientes: 293, 630, 4561, 10274, 12999, 25068, 67554, 163990, estos resultados se encuentran dentro del rango de la complejidad $n \log n$ de cada tamaño del arreglo mencionado.

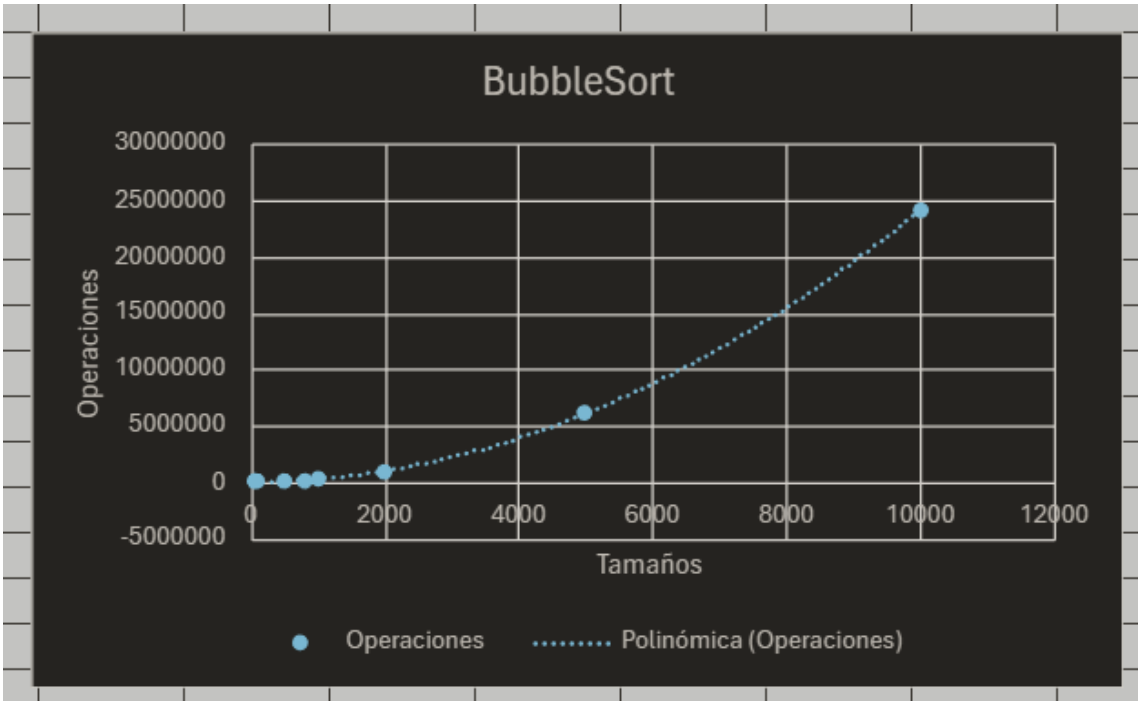


El algoritmo `insertionSort` tiene una complejidad $O(n)$ lineal, si el arreglo se encuentra ordenado ya que la única operación que realizaría serían comparaciones estas serían igual al tamaño del arreglo. Sin embargo, trabajamos con un arreglo desordenado por ello se tiene una complejidad cuadrada para el caso promedio y el peor caso; esto se puede observar en la gráfica donde también se comprobó manualmente y los resultados de las operaciones se encuentran dentro del rango de la complejidad cuadrada usando a cada tamaño del arreglo.

En cuanto a SelectionSort y BubbleSort: EJECUCIONES:


SelectionSort
Operaciones
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1
10
100
1000
10000
Operaciones
Lineal (Operaciones)

O(n2)	Selection Sort
Tamaño	Operaciones
50	45
100	95,4
500	493,4
800	794,2
1000	992
2000	1990,2
5000	4993,2
10000	9988,4


BubbleSort
30000000
25000000
20000000
15000000
10000000
5000000
0
-5000000
Operaciones
0
2000
4000
6000
8000
10000
12000
Tamaños
Operaciones
Polinómica (Operaciones)

O(n)	Bubble Sort
Tamaño	Operaciones
50	605,2
100	2346,2
500	58997,8
800	152450,6
1000	240818,4
2000	966728
5000	6038893,8
10000	24141033

Al verificar los resultados promedio de nuestras gráficas, podemos establecer que:

- **SelectionSort** tiene una complejidad aproximada (dependiendo del mejor o peor caso) de $O(n)$ a $O(n^2)$ que tiende a ser más constante $O(n^2)$ al aumentar el número de elementos del arreglo a ordenar.
- **BubbleSort** tiene una complejidad de $O(n^2)$ que puede reducirse al ser modificada solo al momento de presentarse el mejor caso $O(n)$.

CONCLUSIONES

1. Quetzalli Zarate Menes

Nuestro análisis imparcial nos ha permitido comprender las fortalezas y debilidades de cada algoritmo en diferentes contextos. CountingSort brilla en situaciones donde el rango de valores es amplio, mientras que MergeSort y HeapSort son más eficientes para arreglos más pequeños. La elección del algoritmo adecuado depende de las características específicas del problema y los datos involucrados.

En conclusión, la dispersión de datos es un factor clave al seleccionar el algoritmo de ordenamiento apropiado. CountingSort brilla cuando el rango es pequeño, mientras que MergeSort y HeapSort son más versátiles en una variedad de situaciones.

¿Quien gana QuickSort, HeapSort o MergeSort?

QuickSort es mejor en muchos escenarios prácticos por varias razones:

Constantes pequeñas: En la práctica, las constantes involucradas en la complejidad de QuickSort son menores en comparación con otros algoritmos (O(n log n)), lo que significa que en conjuntos de datos típicos, QuickSort será más rápido.

La superioridad de QuickSort sobre HeapSort puede atribuirse a su mayor eficiencia en la realización de intercambios. QuickSort, mediante su estrategia de partición, asigna a cada elemento su posición definitiva en un solo movimiento, lo que resulta en una menor cantidad de intercambios totales. Por otro lado, HeapSort requiere re-

ajustes constantes del montículo (heap) después de cada eliminación de la raíz, lo que incrementa el número de operaciones necesarias. Además, en el análisis del caso promedio, QuickSort tiende a realizar menos intercambios en comparación con HeapSort, lo que contribuye a su eficacia general y lo convierte en una opción preferente para la ordenación de datos.

2. López Martínez Diana

- El algoritmo `quickSort` demostró tener una complejidad temporal óptima en el mejor caso de $n \log n$, lo cual fue evidenciado tanto por la experimentación como por el análisis manual. Los resultados obtenidos para diferentes tamaños de arreglos se alinearon con la complejidad teórica esperada.
- Para el algoritmo `insertionSort`, se observó que su comportamiento es lineal $O(n)$ cuando el arreglo está previamente ordenado, limitándose a realizar comparaciones. No obstante, en arreglos desordenados, la complejidad se eleva a cuadrática, como se refleja en los resultados experimentales y el análisis manual.
- Las modificaciones realizadas para incluir contadores de operaciones en ambos algoritmos permitieron un análisis detallado de su complejidad. La implementación de un método para reiniciar los contadores resultó ser una solución inicial; sin embargo, la inicialización de los contadores en cero desde el comienzo eliminó la necesidad de reiniciarlos y resolvió el problema de acumulación de operaciones.

3. Giron Escalona Erendira Nayely

Al registrar el número de veces que se realizan operaciones durante la ejecución de un algoritmo, obtenemos una mejor comprensión de su comportamiento en diferentes escenarios y podemos compararlo con otros algoritmos de ordenamiento para determinar su eficacia en términos de tiempo y recursos utilizados.

Los algoritmos de ordenamiento **Selection Sort** y **Bubble Sort** presentan diferentes enfoques y eficiencia en términos de complejidad. **Selection Sort**, aunque simple de entender e implementar, tiene una complejidad cuadrática en el peor de los casos ($O(n^2)$), lo que lo hace menos eficiente para conjuntos de datos grandes. Por otro lado, **Bubble Sort** también tiene una complejidad cuadrática, pero su característica de comparación adyacente puede ser ineficiente, especialmente en arreglos casi ordenados. Ambos algoritmos son adecuados para pequeñas cantidades de datos o como ejemplos introductorios de algoritmos de ordenamiento.

Referencias

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Algoritmos: Teoría y práctica* (3a ed.). McGraw-Hill Interamericana.

[2] Baase, S., & Van Gelder, A. (2002). *Algoritmos computacionales: introducción al análisis y diseño.*

[3] Drozdek, A. (2007). *Estructuras de datos y algoritmos con Java.* Cengage Learning Mexico.

[4] Sierra, F. J. C. (2019). *C/C++: curso de programación.*

[5] Comparación de algoritmos de clasificación. (20 de febrero de 2024). AfterAcademy. `https://afteracademy.com/blog/comparison-of-sorting-algorit`