***Documentación del proyecto***

***Proyecto Java: "El Cáliz de los Códigos Perdidos" Java Developer 20 | Coach: Eduardo Calles |***

***Fecha entrega: 18 mayo 2025***



***Academia: Kodigo***

***Miembros del equipo:***

***Héctor Armando Guevara Paices***

***Raúl Antonio Gallardo Ayala***

***Said Armando Moreno Orellana***

**Introducción**

El proyecto "El Cáliz de los Códigos Perdidos" se basa en una leyenda sobre un oráculo que escondió un mensaje secreto entre miles de números mágicos. La misión es implementar algoritmos de ordenamiento (MergeSort) y búsqueda (búsqueda binaria) para encontrar una clave mágica en un array de 1000 números aleatorios y revelar un mensaje asociado. Los objetivos principales son:

- Implementar MergeSort para ordenar el array.

- Aplicar búsqueda binaria para localizar la clave mágica.

- Analizar la complejidad algorítmica Big O de ambos algoritmos.

- Documentar el código y presentar un informe con resultados de rendimiento y conclusiones

**Explicación de los Algoritmos Implementados**

**Generación de Números Aleatorios y Clave Mágica**

La clase RandomNumbersArray genera un array de números aleatorios y selecciona una clave mágica.

**Método** randomNumbers(int size, int min, int max):

- Genera un array de size números enteros aleatorios entre min (1000) y max (9999).

- Usa Math.random() para generar valores aleatorios.

**Método** magicKey(int[] numbers):

- Selecciona un índice aleatorio del array y retorna el número en esa posición como la clave mágica.

**Algoritmo de Ordenamiento: MergeSort**

La clase MergeSort implementa el algoritmo MergeSort, que divide el array en subarrays, los ordena recursivamente y los fusiona.

**Método** mergeSort(int[] arr, int left, int right):

- Divide el array en dos mitades hasta que cada subarray tiene un solo elemento, luego los fusiona en orden.

**Método** merge(int[] arr, int left, int mid, int right):

- Fusiona dos subarrays ordenados en uno solo.

- Usa arrays temporales para almacenar las mitades y compara elementos para fusionarlos en orden.

**Método** printArray(int[] numbers):

- Imprime los elementos del array (no usado en Main, pero útil para depuración).

**Algoritmo de Búsqueda: BinarySearch**

La clase **BinarySearch** implementa la búsqueda binaria para encontrar la clave mágica en el array ordenado.

**Método** binarySearch(int[] arr, int target):

Busca un valor en un array ordenado dividiendo el espacio de búsqueda a la mitad en cada iteración.

Retorna el índice del elemento si se encuentra, o -1 si no.

**Medición de Tiempo: Timer**

La clase Timer mide el tiempo de ejecución de las operaciones.

**Métodos:**

start(): Inicia el temporizador.

stop(): Detiene el temporizador.

getElapsedTime(): Calcula el tiempo transcurrido en nanosegundos.

**Clase Principal: Main**

La clase Main coordina la ejecución del programa:

- Genera un array de 100 números aleatorios (nota: el proyecto pide 1000, esto podría ser un error).

- Ordena el array con MergeSort.

- Solicita al usuario un número para buscar.

- Usa búsqueda binaria para verificar si el número es la clave mágica.

- Muestra los tiempos de ejecución y complejidades.

**Análisis de Complejidad Big O**

**Generación de Números Aleatorios** (RandomNumbersArray.randomNumbers):

**Complejidad:** O(n), donde n es el tamaño del array (1000). Se itera una vez por cada elemento.

**Selección de Clave Mágica** (RandomNumbersArray.magicKey):

**Complejidad:** O(1). Selecciona un índice aleatorio en tiempo constante.

**MergeSort** (MergeSort.mergeSort y merge):

**Complejidad:** O(n log n). Divide el array en log n niveles, y en cada nivel realiza O(n) operaciones de fusión.

**Justificación:** Es eficiente para arrays grandes y garantiza O(n log n) en todos los casos, a diferencia de QuickSort, que puede ser O(n²) en el peor caso.

**Búsqueda Binaria** (BinarySearch.binarySearch):

**Complejidad:** O(log n). Divide el espacio de búsqueda a la mitad en cada iteración.

**Comparación:** Es mucho más eficiente que la búsqueda lineal (O(n)) para arrays ordenados.

**Medición de Tiempo** (Timer):

**Complejidad:** O(1) para start(), stop(), y getElapsedTime(), ya que solo realizan operaciones básicas.

**Resultados de las pruebas de rendimiento**

**Capturas de pantalla**

Texto

El contenido generado por IA puede ser incorrecto.

Generando un arreglo de números aleatorios...: Corresponde al mensaje inicial en el código que indica que se está creando el arreglo.

Ordenando el arreglo con el algoritmo MergeSort...: Corresponde al mensaje antes de iniciar el ordenamiento.

Ingresa la clave mágica para revelar el mensaje secreto: / 1900 / Qué mala suerte, esa no era la clave mágica, pero puedes seguir intentando.:

Esta parte no está directamente visible en el código Main.java proporcionado.

Esto sugiere fuertemente que hay otra sección del código (o quizás otra clase) que se ejecuta después del ordenamiento y que involucra una búsqueda binaria para un "mensaje secreto" y una clave de entrada por parte del usuario.

El programa está pidiendo al usuario que ingrese un número (en este caso, 95).

Luego, el programa verifica si 95 es la "clave mágica". Como la respuesta es "Qué mala suerte...", significa que 95 no es la clave correcta. Esto implica que la búsqueda binaria se está utilizando para intentar encontrar esa clave dentro del arreglo (probablemente ya ordenado).

--- Análisis de Rendimiento ---: Encabezado que precede a los resultados del rendimiento.

Tiempo que tomó ordenar el arreglo: 56100.0 nanosegundos:

Este es el tiempo que tardó el algoritmo Merge Sort en ordenar el arreglo de 100 elementos.

56100.0 nanosegundos es equivalente a 56.1 microsegundos o 0.0000561 segundos. Es un tiempo muy rápido, lo cual es de esperar para un arreglo de solo 100 elementos.

Complejidad del MergeSort: O(n log n):

Esto es la complejidad temporal teórica del algoritmo Merge Sort. Es una complejidad muy eficiente para algoritmos de ordenamiento, lo que significa que su tiempo de ejecución crece de forma proporcional a n log n, donde n es el número de elementos.

Tiempo que tomó buscar en el arreglo: 504700.0 nanosegundos:

Este es el tiempo que tardó la operación de búsqueda (probablemente la búsqueda binaria mencionada anteriormente).

504700.0 nanosegundos es equivalente a 504.7 microsegundos o 0.0005047 segundos. Es un tiempo un poco más largo que el ordenamiento, lo cual podría deberse a que se realizaron múltiples intentos de búsqueda hasta que el usuario se rindió o a que la búsqueda binaria se aplicó de alguna manera repetidamente, o incluso a que la búsqueda binaria se ejecutó varias veces buscando diferentes "claves mágicas" hasta que se obtuvo una respuesta negativa. Sin el código de la sección de búsqueda, es difícil saber exactamente por qué el tiempo es tan "alto" en comparación con el ordenamiento para un arreglo de 100 elementos.

Complejidad de la búsqueda binaria: O(log n):

Esta es la complejidad temporal teórica del algoritmo de Búsqueda Binaria. Es extremadamente eficiente, ya que el tiempo de ejecución crece logarítmicamente con el número de elementos. Esto significa que para arreglos muy grandes, la búsqueda binaria sigue siendo increíblemente rápida.

Texto

El contenido generado por IA puede ser incorrecto.

Importaciones:

import Utils.BinarySearch;: Importa una clase BinarySearch (presumiblemente ubicada en el paquete Utils) que contiene el algoritmo de búsqueda binaria.

import Utils.MergeSort;: Importa una clase MergeSort (en Utils) que implementa el algoritmo de ordenamiento Merge Sort.

import Utils.RandomNumbersArray;: Importa una clase RandomNumbersArray (en Utils) que probablemente se encarga de generar arreglos de números aleatorios.

import Utils.Timer;: Importa una clase Timer (en Utils) que se utiliza para medir el tiempo de ejecución de las operaciones.

Clase Main:

public class Main { ... }: Esta es la clase principal donde se ejecuta el programa.

Método main:

public static void main(String[] args) { ... }: El punto de entrada del programa.

Inicialización de Timer y MergeSort:

Timer temporizadorOrdenamiento = new Timer();: Crea una instancia de Timer para medir el tiempo de ordenamiento.

Timer temporizadorBusqueda = new Timer();: Crea una instancia de Timer para medir el tiempo de búsqueda.

MergeSort merge = new MergeSort();: Crea una instancia de la clase MergeSort.

Generación de un arreglo de números aleatorios:

System.out.println("Generando un arreglo de números aleatorios...");: Mensaje en consola.

int[] numeros = RandomNumbersArray.RandomNumbers(size: 100, min: 1000, max: 9999);: Llama a un método estático RandomNumbers de RandomNumbersArray para crear un arreglo de 100 enteros, con valores entre 1000 y 9999 (inclusive).

Ordenamiento con MergeSort:

System.out.println("Ordenando el arreglo con el algoritmo MergeSort...");: Mensaje en consola.

temporizadorOrdenamiento.start();: Inicia el temporizador para el ordenamiento.

MergeSort.mergeSort(numeros, left: 0, right: numeros.length - 1);: Llama al método estático mergeSort de la clase MergeSort para ordenar el arreglo numeros. Se le pasa el arreglo, el índice izquierdo (0) y el índice derecho (último elemento).

temporizadorOrdenamiento.stop();: Detiene el temporizador.

Obtención del tiempo de ordenamiento:

double tiempoOrdenamiento = temporizadorOrdenamiento.getElapsedTime();: Obtiene el tiempo transcurrido en nanosegundos (probablemente, dado el output) desde el temporizador.

Comentario para imprimir el arreglo ordenado:

// merge.printArray(numeros);: Esta línea está comentada, lo que significa que el programa no imprime el arreglo ordenado a menos que se descomente.

**Texto

El contenido generado por IA puede ser incorrecto.**

Medición del tiempo de búsqueda:

temporizadorBusqueda.start();: Inicia el temporizador específico para la búsqueda.

int resultado = BinarySearch.binarySearch(numeros, entradaUsuario);: Aquí es donde ocurre la magia. Se llama al método estático binarySearch de la clase BinarySearch.

numeros: Es el arreglo que ya ha sido ordenado por Merge Sort. La búsqueda binaria requiere que el arreglo esté ordenado para funcionar correctamente.

entradaUsuario: Esta variable (que no se ve en este fragmento, pero es inferida por la interacción de la consola) contendría el valor que el usuario ingresó por teclado (en el ejemplo anterior, 95).

El método binarySearch devuelve el índice del elemento si lo encuentra, o un valor especial (comúnmente -1) si no lo encuentra.

temporizadorBusqueda.stop();: Detiene el temporizador de búsqueda.

double tiempoBusqueda = temporizadorBusqueda.getElapsedTime();: Obtiene el tiempo transcurrido para la operación de búsqueda.

Verificación del resultado de la búsqueda:

if (resultado == -1) { ... }:

Si resultado es -1, significa que la entradaUsuario (la clave ingresada) no fue encontrada en el arreglo numeros.

System.out.println("Qué mala suerte, esa no era la clave mágica, pero puedes seguir intentando.");: Se imprime el mensaje que vimos en la ejecución, indicando que la clave no es correcta.

else { ... }:

Si resultado no es -1, significa que la entradaUsuario fue encontrada en el arreglo, y resultado es el índice donde se encontró.

System.out.println("¡Has encontrado la clave mágica: " + numeros[resultado] + " Eres el elegido!");: Se imprime un mensaje de éxito, revelando la "clave mágica" (que es el número encontrado en el índice resultado del arreglo numeros).

Impresión del Análisis de Rendimiento:

System.out.println("\n--- Análisis de Rendimiento ---");: Imprime un encabezado para la sección de rendimiento.

System.out.println("Tiempo que tomó ordenar el arreglo: " + tiempoOrdenamiento + " nanosegundos"): Muestra el tiempo calculado para Merge Sort.

System.out.println("Complejidad del MergeSort: O(n log n)"): Reafirma la complejidad teórica de Merge Sort.

System.out.println("Tiempo que tomó buscar en el arreglo: " + tiempoBusqueda + " nanosegundos"): Muestra el tiempo calculado para la Búsqueda Binaria.

System.out.println("Complejidad de la búsqueda binaria: O(log n)"): Reafirma la complejidad teórica de la Búsqueda Binaria.

**Conclusiones**

**Rendimiento**: MergeSort es eficiente para ordenar el array de 1000 elementos, con una complejidad de O(n log n). La búsqueda binaria es extremadamente rápida (O(log n)), ideal para arrays ordenados.

**Aplicaciones Prácticas:**

- MergeSort es útil en sistemas donde se necesita un ordenamiento estable

- Búsqueda binaria es ideal para buscar en colecciones grandes y ordenadas

**Desafíos:**

- Asegurar que el array esté ordenado antes de la búsqueda binaria.

- Seleccionar una clave mágica y asociar un mensaje secreto de forma clara.

**Referencias**

- Cormen, T. H., et al. (2009). Introduction to Algorithms. MIT Press.

- Documentación oficial de Java: <https://docs.oracle.com/en/java/>

- Tutoriales de MergeSort y BinarySearch: GeeksforGeeks.

**Estructura del proyecto**

El\_Caliz\_De\_Los\_Codigos\_Perdidos/

├── src/

│ ├── Main.java

│ └── Utils/

│ ├── BinarySearch.java

│ ├── MergeSort.java

│ ├── RandomNumbersArray.java

│ └── Timer.java

├── README.md