Java Developer 20

El Cáliz de los Códigos Perdidos

Desarrolladores:

Cristina Raquel Alfaro Aguilar

Judith Esther Arévalo Guardado

Erick Alexander Montoya Cruz

Coach:

Eduardo Calles

Introducción

El Cáliz de los Códigos Perdidos se basa en una historia de fantasía que plantea un reto técnico: encontrar un mensaje secreto oculto entre miles de números mágicos mediante el uso de algoritmos de ordenamiento, búsqueda y análisis de rendimiento algorítmico. La solución fue implementada con los conocimientos previamente obtenidos en las clases Java Developer 20.

Objetivos

- Aplicar QuickSort o MergeSort para ordenar números.
- Usar búsqueda binaria para encontrar la clave mágica.
- Evaluar la complejidad algorítmica y el rendimiento de cada uno.
- Documentar los pasos, estructuras y decisiones del proyecto.

Estructura del proyecto

src/

- **Main.java** → Punto de entrada
- **GeneradorNumeros.java** → Generación y selección de la clave
- **QuickSort.java** → Algoritmo de ordenamiento
- **MergeSort.java** → Algoritmo de ordenamiento op 2
- **BusquedaBinaria.java** → Algoritmo de búsqueda binaria
- **Mensajes.java** → Mensajes secretos aleatorios

Algoritmos utilizados

QuickSort

Divide y conquista.

Elige un pivote, separa menores y mayores.

Se aplica recursivamente.

Complejidad promedio: O(n log n)

Peor caso: $O(n^2)$

MergeSort

Divide y conquista.

Divide el arreglo en mitades hasta llegar a subarreglos de 1 elemento.

Luego fusiona los subarreglos ordenadamente.

Se aplica recursivamente hasta unir todo el arreglo.

Complejidad promedio y peor caso: O(n log n)

Uso óptimo: cuando se busca consistencia en el rendimiento, ya que no empeora con el tipo de datos.

Búsqueda binaria

Requiere arreglo ordenado.

Divide el rango en mitades.

Eficiente en tiempo: O(log n)

Ejemplo de códigos:

QuickSort

```java
public class QuickSort {  2 usages  judsther *

    public static void quickSort(int[] arr, int izquierda, int derecha){
        if(izquierda < derecha) {
            int indiceParticion = particion(arr, izquierda, derecha);

            quickSort(arr, izquierda,  derecha: indiceParticion - 1);
            quickSort(arr,  izquierda: indiceParticion + 1, derecha);
        }
    }

    private static int particion(int[] arr, int izquierda, int derecha){
        int pivote = arr[derecha];
        int i = izquierda - 1;

        for (int j = izquierda; j < derecha; j++){
            if(arr[j] < pivote){
                i++;

                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[derecha];
        arr[derecha] = temp;
    //retornamos el indice de particion
        return i + 1;
    }
}
```

MergeSort

```
public class MergeSort {

    // Método principal que el usuario puede usar
    public static void ordenar(int[] arreglo) {
        if (arreglo == null || arreglo.length < 2) return;
        mergeSort(arreglo, 0, arreglo.length - 1);
    }

    private static void mergeSort(int[] arreglo, int izquierda, int derecha) {
        if (izquierda < derecha) {
            int medio = (izquierda + derecha) / 2;

            // Dividir recursivamente
            mergeSort(arreglo, izquierda, medio);
            mergeSort(arreglo, medio + 1, derecha);

            // Combinar
            merge(arreglo, izquierda, medio, derecha);
        }
    }

    private static void merge(int[] arreglo, int izquierda, int medio, int derecha) {
        int n1 = medio - izquierda + 1;
        int n2 = derecha - medio;

        int[] izquierdaTemp = new int[n1];
        int[] derechaTemp = new int[n2];

        // Copiar datos temporales
        for (int i = 0; i < n1; ++i)
            izquierdaTemp[i] = arreglo[izquierda + i];
        for (int j = 0; j < n2; ++j)
            derechaTemp[j] = arreglo[medio + 1 + j];
```

MergeSort (continuación)

```
        // Mezclar arreglos temporales
            int i = 0, j = 0, k = izquierda;

            while (i < n1 && j < n2) {
                if (izquierdaTemp[i] <= derechaTemp[j]) {
                    arreglo[k] = izquierdaTemp[i];
                    i++;
                } else {
                    arreglo[k] = derechaTemp[j];
                    j++;
                }
                k++;
            }

            // Copiar los elementos restantes
            while (i < n1) {
                arreglo[k] = izquierdaTemp[i];
                i++;
                k++;
            }
            while (j < n2) {
                arreglo[k] = derechaTemp[j];
                j++;
                k++;
            }
        }
    }
```

BinarySearch

```java
public class BusquedaBinaria {  2 usages  judsther
    /**
     * Realizamos la búsqueda binaria en nuestro arreglo ordenado
     * @param arr - Arreglo ordenado de enteros
     * @param clave - Número que se quiere buscar
     * @return true si se encuentra la clave, false si no
     *
     * Complejidad algorítmica: O(log n)  porque en cada paso se descarta
     */

    public static boolean buscar(int[] arr, int clave){  1 usage  judsther
        int izquierda = 0;
        int derecha = arr.length - 1;

        while (izquierda <= derecha){
            //localizar el indice del medio
            int medio = (izquierda + derecha) / 2;

            //buscamos la clave primero comparamos con el valor del indic
            if(arr[medio] == clave){
                return true;
            } else if (arr[medio] < clave ){
                izquierda = medio + 1;
            } else {
                derecha = medio - 1;
            }
        }
    return false;
    }
```

Descripcón del flujo:

1. Se genera un array de 1000 números entre 1000 y 9999.
2. Se elige una clave mágica aleatoria.
3. Se ordena el array con QuickSort o MergeSort .
4. El usuario intenta adivinar el número mágico.
5. Se aplica búsqueda binaria.
6. Si acierta, se revela un mensaje secreto, si no acierta también mostramos un mensaje distinto.
7. Se mide y reporta el tiempo de ordenamiento y búsqueda.

Resultados de las pruebas de rendimiento

QuickSort - MergeSort - BinarySearch

Algoritmo	Complejidad promedio	Peor caso	Estable	En memoria	Tiempo
QuickSort	O(n log n)	$O(n^2)$	No	In-place	0.000623 s
MergeSort	O(n log n)	O(n log n)	Sí	Requiere adicional	0.000916 s
BinarySearch	O(log n)	O(log n)	Sí	In-place	0.000315 s

Conclusiones

Rendimiento relativo de los algoritmos y sus aplicaciones prácticas.

- QuickSort es muy efectivo en conjuntos medianos y ofrece gran rendimiento en promedio, aunque su peor caso puede degradarse si no se elige bien el pivote.
- MergeSort ofrece un rendimiento estable y predecible, ideal cuando se requiere consistencia en los tiempos, aunque usa más memoria.
- La búsqueda binaria es extremadamente eficiente una vez que el arreglo está ordenado, mostrando tiempos muy bajos incluso con grandes volúmenes de datos.
- El análisis de tiempos de ejecución permitió comparar ambos algoritmos en la práctica, confirmando la teoría de complejidad algorítmica.

Recomendaciones

- Siempre ordenar el arreglo antes de aplicar búsqueda binaria.
- Usar QuickSort cuando se necesite rapidez en promedio y no preocupe el uso exacto de memoria.
- Usar MergeSort cuando requiera estabilidad en el rendimiento y se estés dispuesto a usar un poco más de memoria.
- Si se usa QuickSort en proyectos reales, debe considerarse la aleatorización del pivote para evitar el peor caso en arreglos casi ordenados.
- Analizar el contexto del problema para elegir el algoritmo más adecuado, no solo por velocidad, sino también por consistencia, memoria y tipo de datos