Guatemala 28 de enero de 2024
Facultad de Ingeniería
Algoritmos y Estructura de Datos
Vianka Castro 23201
Ricardo Godinez 23247

Hoja de Trabajo 3
Diseño del ADT para pilas (stack)

Repositorio

Enlace al Repositorio

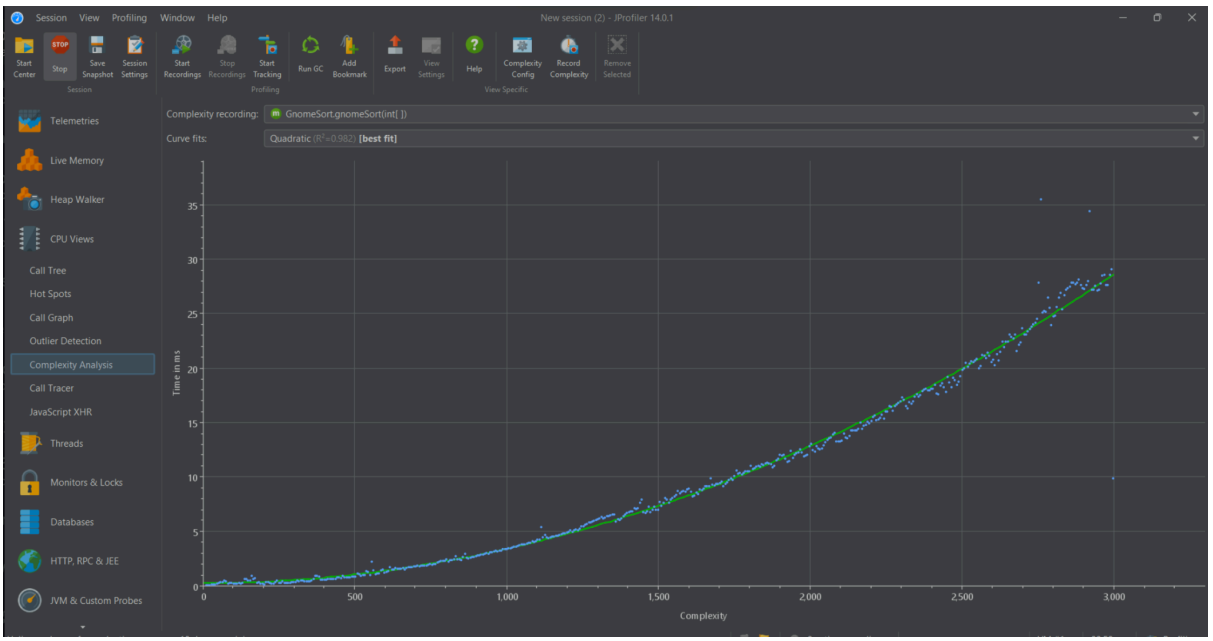
Gnome sort

Este algoritmo de ordenamiento es bastante simple y se empleó iterando sobre la lista, comparando elementos adyacentes y moviéndolos si están en el orden incorrecto, similar al algoritmo de burbuja.

Este es un algoritmo más simple y menos eficiente, con una complejidad de tiempo promedio y en el peor caso de $O(n^2)$, n es el número de elementos. Sin embargo, el peor caso puede ser evitado en la práctica con una buena elección del pivote, haciendo de QuickSort uno de los algoritmos de ordenamiento más rápidos para uso general.

- Inicialmente, se cargó el ArrayList con 3000 números al azar.
- Se itera sobre la lista comparando cada elemento con su predecesor.
- Si el elemento actual es mayor que su predecesor, se avanza al siguiente elemento.
- Si el elemento actual es menor o igual que su predecesor, se intercambia los elementos y retrocedes un índice en la lista.
- Se repitió este proceso hasta que el elemento actual esté en la posición correcta.
- Una vez que haya recorrido toda la lista, esta estará ordenada.

Resultados



En los resultados se puede observar que entre más tiempo concurría, más la complejidad subía. Esto nos demuestra que el Gnome Sort contenía un alto nivel de complejidad al transcurso de la corrida de números.

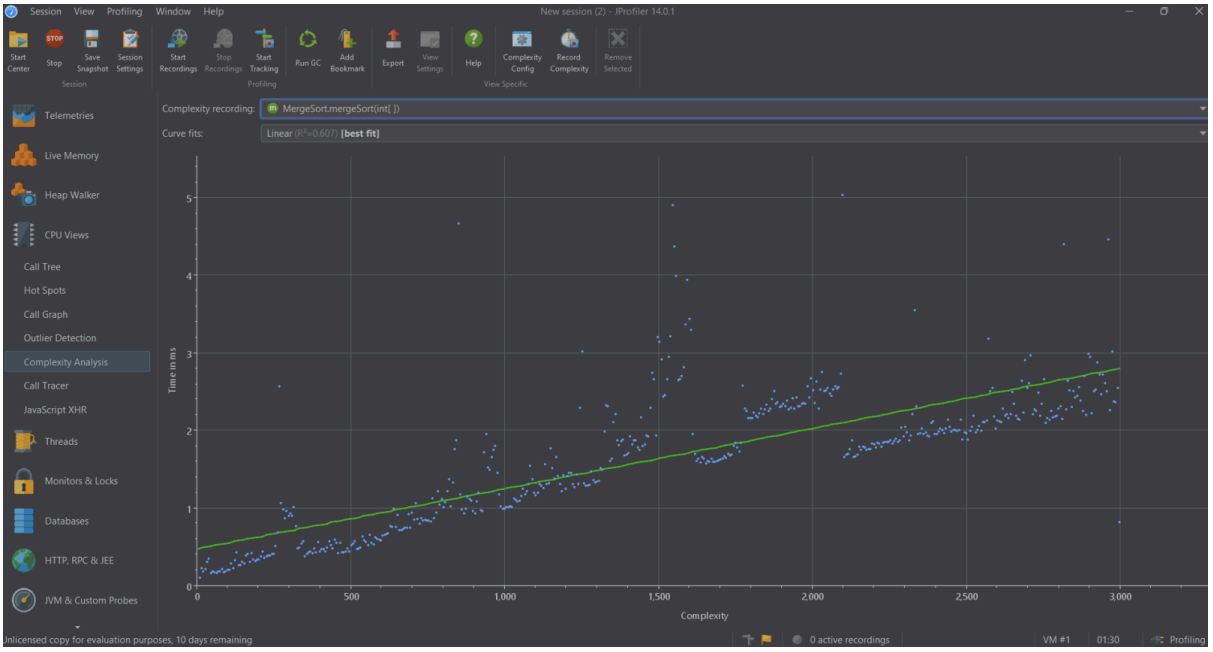
Merge sort O(nlogn)

Merge Sort es un algoritmo de ordenamiento basado en la técnica divide y conquista. Divide la lista en mitades, ordena cada mitad de manera recursiva y luego combina las mitades ordenadas. Para evaluar su rendimiento, un profiler podría registrar el tiempo de ejecución y el uso de memoria, ya que Mergesort es eficiente en términos de tiempo, pero puede requerir una cantidad significativa de memoria adicional.

Merge Sort es un algoritmo de división y conquista con una complejidad de tiempo constante de O(nlogn) en el peor, mejor, y caso promedio, siendo n el número de elementos en el array.

- Cargamos el ArrayList con 3000 números al azar.
- Llamamos a la función o método que implementa el algoritmo de Mergesort, pasando el ArrayList como parámetro.
- El algoritmo Merge Sort ordenará la lista y, una vez que finalice la ejecución, se tendrá la lista ordenada.

Resultados



En este caso, MergeSort tiene una variación alta entre su complejidad y el tiempo transcurrido. Se pueden observar ciertos puntos independientes lo que podría demostrar una falla en el empleo o una mala distribución.

Quick sort

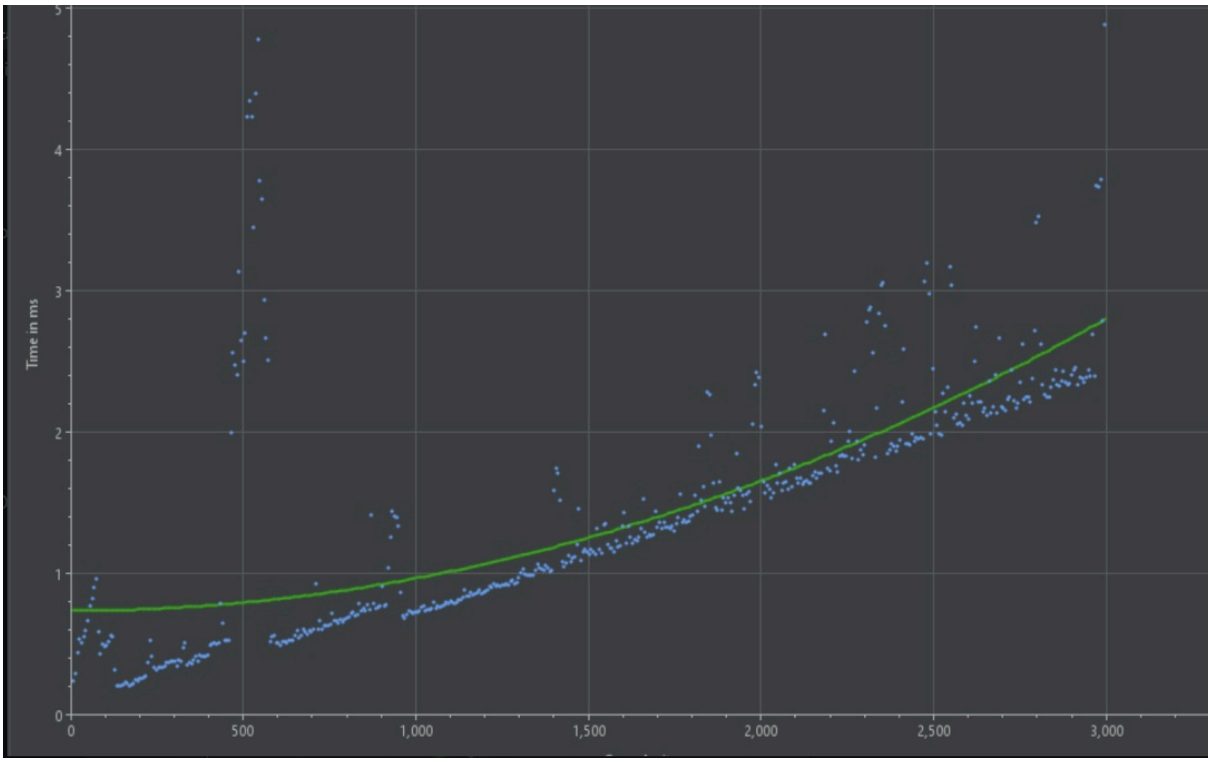
Quick Sort es un algoritmo de ordenamiento basado en divide y conquista que selecciona un elemento como pivote y participa la lista alrededor del pivote. Es muy eficiente en la práctica y generalmente supera a otros algoritmos de ordenamiento en la mayoría de los casos. Para evaluar su rendimiento, podrías utilizar un profiler que registre el tiempo de ejecución, el número de comparaciones y el número de intercambios realizados.

Tiene una complejidad de tiempo en el caso promedio de O(nlogn), pero su peor caso es O(n^2). Sin embargo, el peor caso puede ser evitado con una buena elección del pivote, haciendo este sort uno de los algoritmos de ordenamiento más rápidos para uso general.

- Seleccionamos un elemento como pivote (generalmente el primer elemento de la lista).

- Ordenamos los elementos de la lista de manera que todos los elementos menores que el pivote estén a su izquierda y todos los elementos mayores estén a su derecha.
- Recursivamente, aplicamos el mismo proceso a las sublistas generadas a la izquierda y a la derecha del pivote hasta que la lista esté completamente ordenada.

Resultados



La complejidad con respecto al tiempo en esta gráfica refleja un incremento considerablemente constante mientras corre el programa. Aunque existen algunos pivotes destacables al principio, la gráfica se mantiene uniforme a comparación de otros tipos de sorteo siendo bastante rápido y eficiente.

Radix Sort O(nk)

Radix Sort es un algoritmo de ordenamiento no comparativo que ordena los elementos procesando los dígitos individuales de los números. Es especialmente útil para ordenar números enteros en representación binaria. Para evaluar su rendimiento, podrías utilizar un profiler que registre el tiempo de ejecución y la cantidad de operaciones realizadas en cada paso del algoritmo.

La complejidad de tiempo de Radix Sort depende del número de dígitos (n) de los números a ordenar y del rango de estos dígitos (k). Generalmente, se considera como O(nk) para la mayoría de las implementaciones. Sin embargo, en contextos donde k es relativamente pequeño comparado con n, se puede considerar casi lineal.

- Ordenamos los elementos procesando los dígitos individuales de los números.
- Iniciamos con el dígito menos significativo y ordenamos los elementos en función de este dígito.
- Luego, se repite el proceso para el siguiente dígito más significativo, y así sucesivamente, hasta que se hayan considerado todos los dígitos.

Resultados

Insertion Sort

Insertion Sort es un algoritmo de ordenamiento simple que construye una lista ordenada una inserción a la vez, moviendo elementos hacia la posición correcta. Aunque no es tan eficiente como otros algoritmos en listas grandes, puede ser más rápido en listas pequeñas o listas que ya están parcialmente ordenadas. Para evaluar su rendimiento, un

profiler podría registrar el tiempo de ejecución y el número de comparaciones y movimientos de elementos.

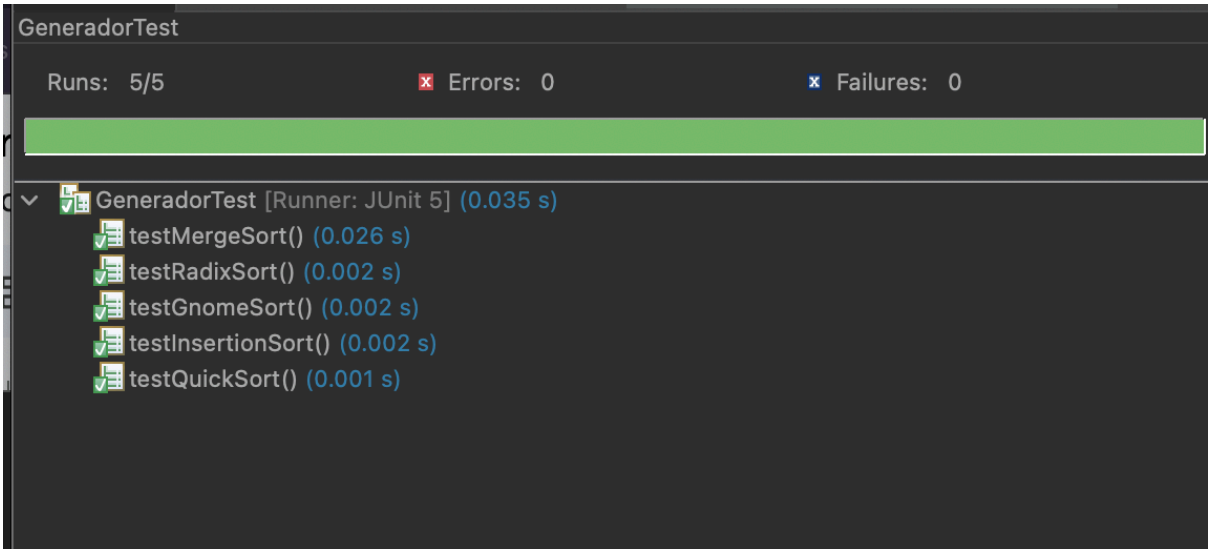
Este algoritmo tiene una complejidad de tiempo en el peor caso y en el caso promedio de O(n^2), donde n es el número de elementos. En el mejor caso, cuando el arreglo ya está ordenado, su complejidad es O(n).

- Comienza con una lista desordenada y toma un elemento de la lista.
- Inserta este elemento en la posición correcta en la parte ordenada de la lista, desplazando los elementos mayores que él hacia la derecha.
- Repite este proceso hasta que todos los elementos estén en su posición correcta.

Resultados

Por último, en InsertionSort al ser de ordenamiento simple, se puede ver un nivel medio de dificultad iniciando de manera menor hasta llegar a valores de complejidad bastantes altos mientras transcurre el tiempo. Al ser una lista muy grande, también proyecta una complejidad alta y poca eficiencia.

Pruebas de JUnit



```java
import static org.junit.jupiter.api.Assertions.*;

class GeneradorTest {

    @Test
    public void testGnomeSort() {

        int[] arr = {5, 2, 7, 3, 9, 1};
        int[] expected = {1, 2, 3, 5, 7, 9};

        GnomeSort.gnomeSort(arr);

        assertArrayEquals(expected, arr);
    }

    @Test
    public void testMergeSort() {
        // Arrange
        int[] arr = {5, 2, 7, 3, 9, 1};
        int[] expected = {1, 2, 3, 5, 7, 9};

        // Act
        MergeSort.mergeSort(arr);

        // Assert
        assertArrayEquals(expected, arr);
    }


    @Test
    public void testQuickSort() {
        // Arrange
        int[] arr = {5, 2, 7, 3, 9, 1};
        int[] expected = {1, 2, 3, 5, 7, 9};

        // Act
        QuickSort.quickSort(arr, 0, arr.length - 1);

        // Assert
        assertArrayEquals(expected, arr);


    }

     @Test
        public void testRadixSort() {
            // Arrange
            int[] arr = {5, 2, 7, 3, 9, 1};
            int[] expected = {1, 2, 3, 5, 7, 9};

            // Act
            RadixSort.radixSort(arr, arr.length);

            // Assert
            assertArrayEquals(expected, arr);
        }
```