

Sharis Barrios García
Leonel Contreras Quirós
Allan Paniagua Enríquez

Hoja de Trabajo 3

Enlace de repositorio de Git: <https://github.com/sharlisbg/HDT-3>

En este enlace se deja evidencia de todo el desarrollo realizado.

PROFILER UTILIZADO

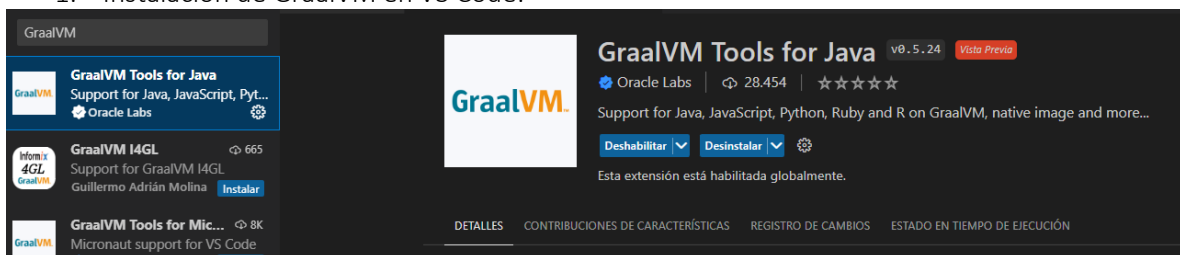
Para VS Code se instaló GraalVM que permite una integración con VisualVM, el cual es el profiler a utilizar para conocer el rendimiento en tiempo de nuestros algoritmos de ordenamiento. Además, este profiler es la herramienta de solución de problemas y monitoreo de Java (y polígota) todo en uno.

Fuentes:

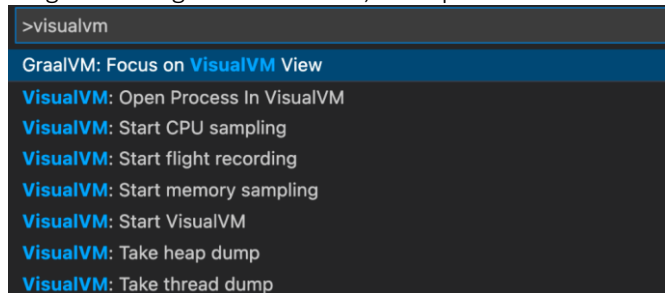
- <https://marketplace.visualstudio.com/items?itemName=oracle-labs-graalvm.graalvm>
- <https://www.graalvm.org/dev/tools/vscode/graalvm-extension/visualvm-integration/>

Paso a Paso de Instalación para uso:

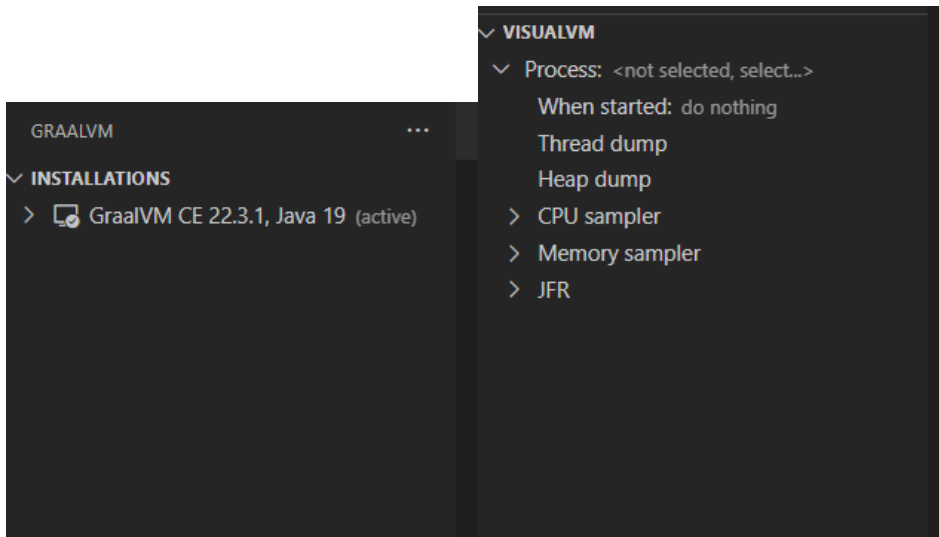
1. Instalación de GraalVM en VS Code:



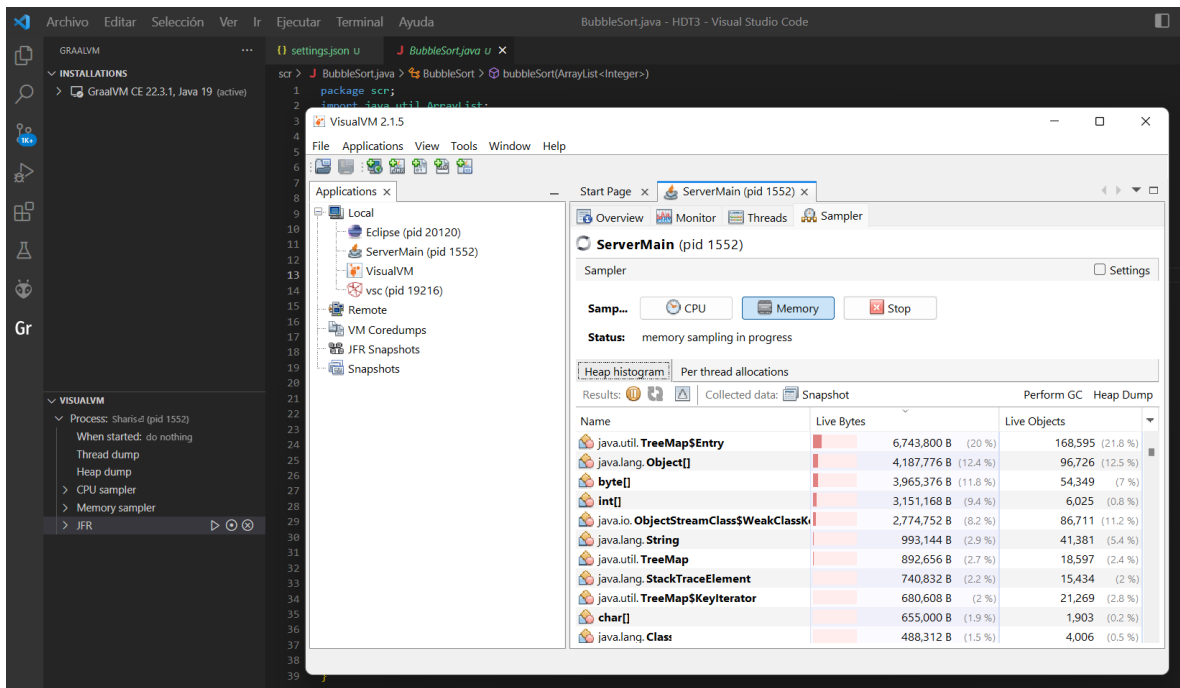
2. Instalación de VisualVM: Primeramente, se descarga el programa en la computadora y luego se configura en VS Code, en la parte de Paleta de Comandos, el siguiente entorno:



- Posteriormente se configuran los elementos expuestos y ya aparecen las siguientes opciones:



- Luego podemos darle Play a JFR y ya estaremos utilizando el profiler para conocer en tiempo real la ejecución de nuestros algoritmos.



RESULTADOS – COMPARACIÓN

Los algoritmos de ordenamiento utilizados fueron los siguientes:

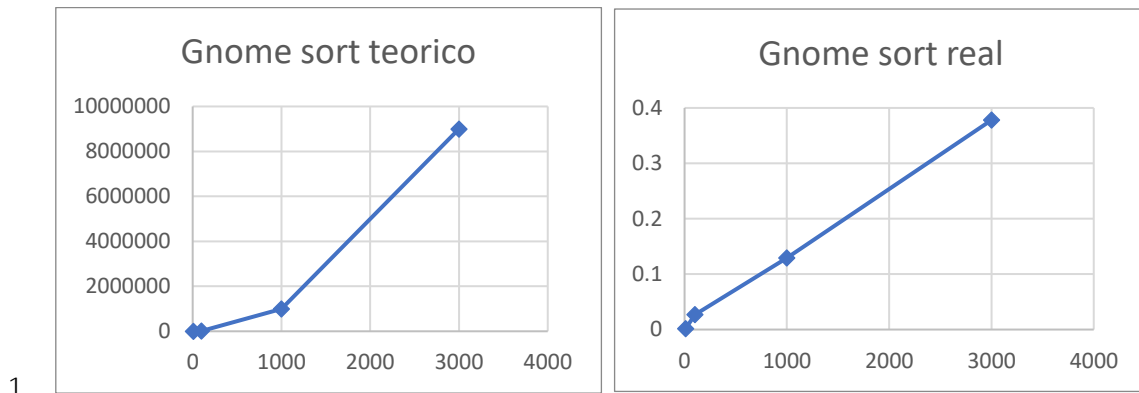
- Gnome sort: <https://www.geeksforgeeks.org/java-program-for-gnome-sort/>
- Merge sort- Fuente: <https://www.baeldung.com/java-merge-sort#:~:text=Merge%20sort%20is%20a%20%E2%80%9Cdivide,final%20solution%20to%20the%20problem.>
- Quick sort: Ejercicio creado en clase.
- Selection Sort: Desarrollado por el equipo.
- Bubble Sort: Desarrollado por el equipo.

Este último algoritmo de ordenamiento fue el elegido en el grupo para ser trabajado.

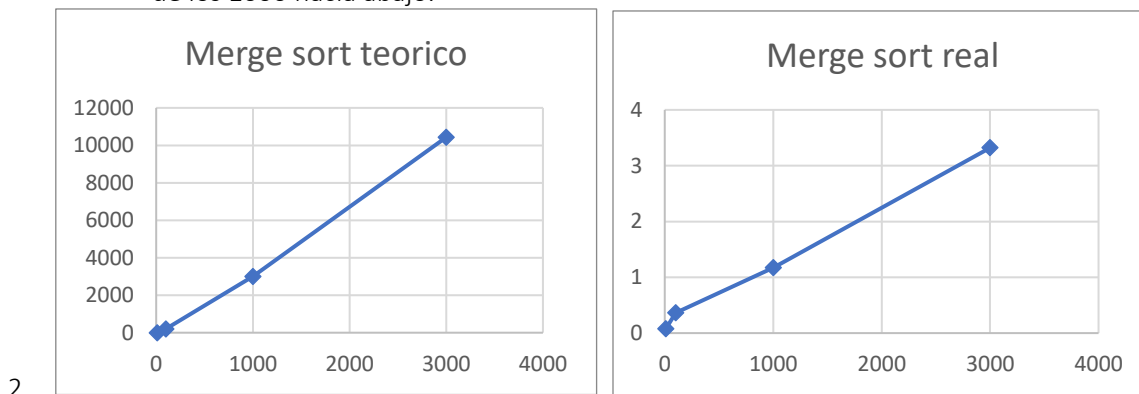
El ejercicio constará en realizar 4 sets de corridas para ordenar, 10, 100, 1000 y 3000 números generados de manera aleatoria. A continuación, se detallan los datos obtenidos y las evidencias de los casos:

Prueba	Cantidad de datos		Teórico	Real
<i>Gnome sort</i> $O(n^2)$	10	100		0.0015
	100	10,000		0.027
	1000	1,000,000		0.129
	3000	9,000,000		0.378
<i>Merge sort</i> $O(n \log n)$	10	10		0.078
	100	200		0.362
	1000	3,000		1.17
	3000	10,431		3.32
<i>Quick sort</i> $O(n \log n)$	10	10		0.073
	100	200		1.53
	1000	3,000		8.34
	3000	10,431		19.5
<i>Selection sort</i> $O(n^2)$	10	10		0.052
	100	100		0.502
	1000	1,000		13.1
	3000	3,000		27.9
<i>Bubble sort</i> $O(n^2)$	10	100		0.204
	100	10,000		0.739
	1000	1,000,000		23.4
	3000	9,000,000		98.9

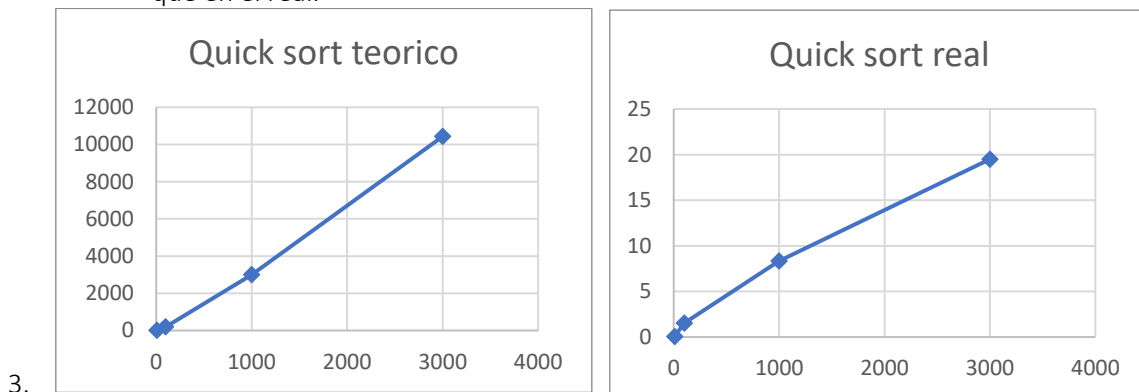
Las gráficas que evidencian los tiempos de corrida teóricos vs teóricos de los algoritmos según las diversas escalas son los siguientes



- a. Se puede observar en ambas graficas como a medida que los datos aumentan el tiempo aumenta con ellos. Como se puede observar los comportamientos difieren de los 1000 hacia abajo.

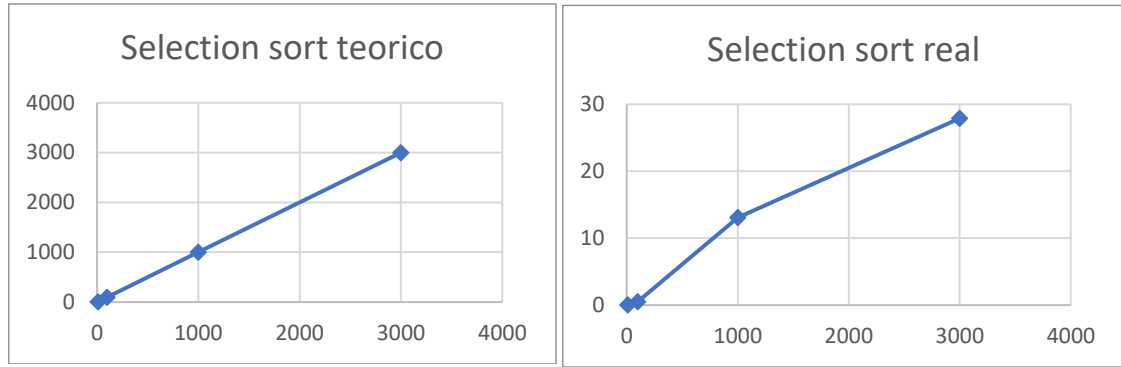


- a. En estas graficas podemos observar que los comportamientos difieren un del otro, ya que el aumento de los tiempos respecto a los datos en el teórico es mas abrupto que en el real.



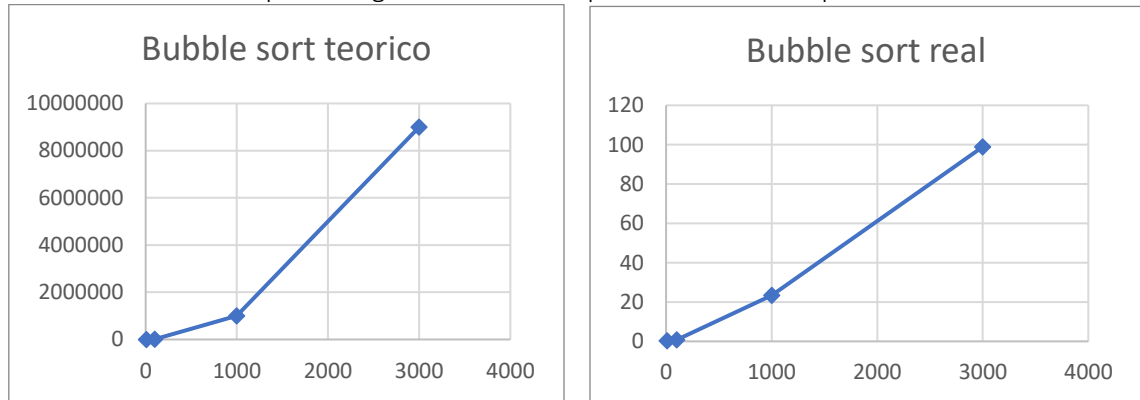
- a. En esta grafica cambia completamente el comportamiento de las graficas ya que en el teórico se ve un comportamiento más lineal, pudiendo tomar como una ligera curva a una grafica exponencial mientras en el real se va a semejando a una grafica logarítmica.

4.



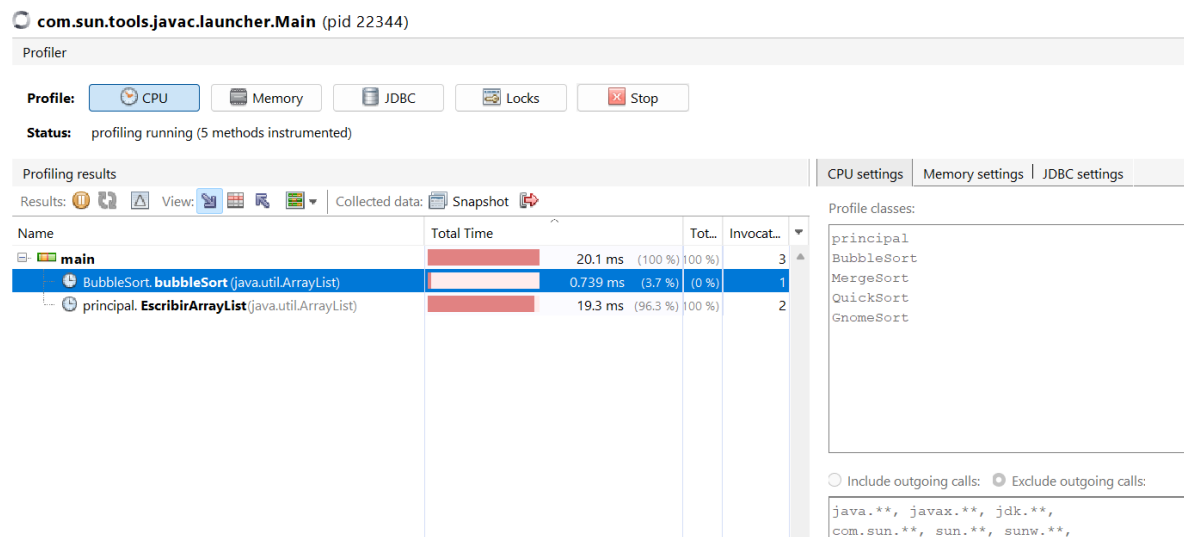
- a. En esta se puede ver un comportamiento similar a la gráfica de quick sort, con el distintivo de que en la grafica de los tiempos teóricos es completamente lineal.

5.



- a. En estas graficas se puede observar que se tiene una tendencia a un tipo de grafica exponencial, siendo de mayor crecimiento, respecto datos-tiempo, la gráfica teórica a diferencia de la grafica real ya que en esta segunda se puede observar un crecimiento más lento del tiempo de ordenamiento.

Evidencias:





Universidad del Valle de Guatemala

Algoritmos y Estructura de Datos

File Applications View Tools Window Help

com.sun.tools.javac.launcher.Main (pid 22520) x com.sun.tools.javac.launcher.Main (pid 22344) x

Overview Monitor Threads Sampler Profiler

com.sun.tools.javac.launcher.Main (pid 22344)

Profiler

Profile: CPU Memory JDBC Locks Stop

Status: profiling running (5 methods instrumented)

Profiling results

Results: View: Collected data: Snapshot

Name	Total Time	Tot..	Invocat..
main	934 ms (100 %)	100 %	9
BubbleSort.bubbleSort (java.util.ArrayList)	59.5 ms (6.4 %)	6.4 %	3
principal.EscribirArrayList (java.util.ArrayList)	875 ms (93.6 %)	3.6 %	6

Profile classes:

principal
BubbleSort
MergeSort
QuickSort
GnomeSort

Include outgoing calls: Exclude outgoing calls:

java.**, javax.**, jdk.**,
com.sun.**, sun.**, sunw.**,
apple.laf.**, apple.awt.**, com.apple.**,
org.omg.CORBA.**, org.omg.CosNaming.**, com.rsa.**

Preset: Custom

com.sun.tools.javac.launcher.Main (pid 22344)

Profiler

Profile: CPU Memory JDBC Locks Stop

Status: profiling running (10 methods instrumented)

Profiling results

Results: View: Collected data: Snapshot

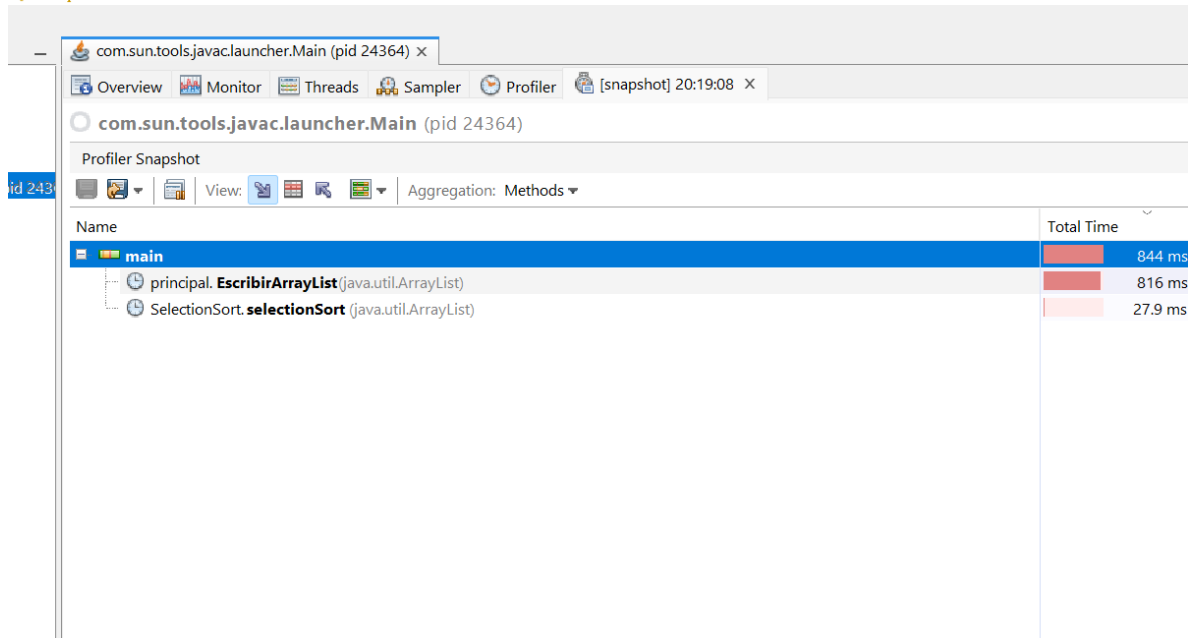
Name	Total Time	Total Time (CPU)	Invocations
main	1,731 ms (100 %)	327 ms (100 %)	27
principal.EscribirArreglo (int[])	946 ms (54.7 %)	124 ms (38.2 %)	10
principal.EscribirArrayList (java.util.ArrayList)	762 ms (44 %)	187 ms (57.3 %)	8
QuickSort.quickSort (int[], int, int, java.util.Comparator)	21.7 ms (1.3 %)	14.7 ms (4.5 %)	4
QuickSort.quickSort (int[], int, int, java.util.Comparator)	19.5 ms (1.1 %)	15.7 ms (4.8 %)	8
ComparadorEnteros.compare (Object, Object)	1.24 ms (0.1 %)	0.0 ms (0 %)	4,114
Self time	0.877 ms (0.1 %)	0.0 ms (0 %)	4
GnomeSort.gnomeSort (java.util.ArrayList)	0.378 ms (0 %)	0.0 ms (0 %)	4
MergeSort.mergeSort (int[], int)	0.101 ms (0 %)	0.0 ms (0 %)	1
MergeSort.mergeSort (int[], int)	0.078 ms (0 %)	0.0 ms (0 %)	2
Self time	0.022 ms (0 %)	0.0 ms (0 %)	1
MergeSort.merge (int[], int[], int[], int, int)	0.0 ms (0 %)	0.0 ms (0 %)	1

Profile classes:

principal
BubbleSort
MergeSort
QuickSort
GnomeSort
RadixSort

Include outgoing calls: Exclude outgoing calls:

java.**, javax.**, jdk.**,
com.sun.**, sun.**, sunw.**,
apple.laf.**, apple.awt.**,
org.omg.CORBA.**, org.omg.CosNaming.**, com.rsa.**



PRUEBAS UNITARIAS

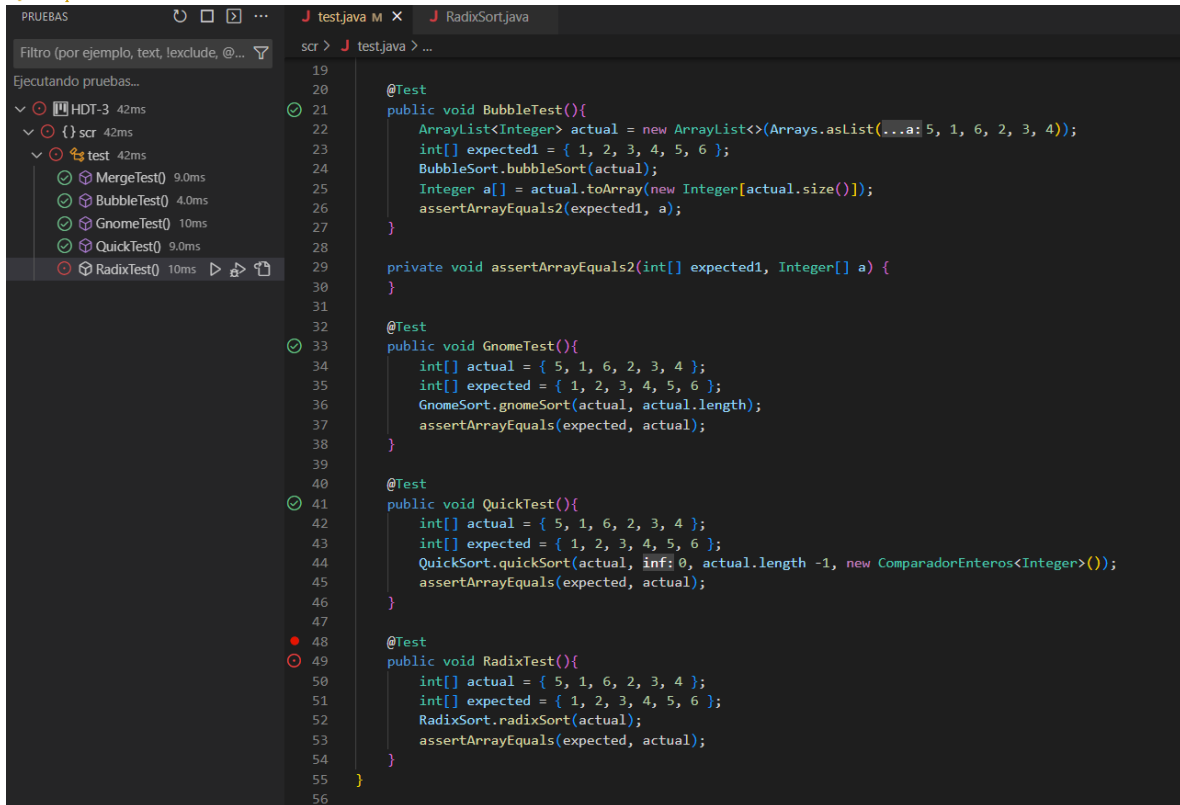
Se ejecutaron las siguientes pruebas unitarias para evidenciar que los algoritmos de ordenamiento tienen un correcto funcionamiento.

0/0 pruebas superadas (0.00 %)

```

1
2
3  import static org.junit.Assert.assertEquals;
4  import org.junit.Test;
5
6
7  public class test {
8
9      @Test
10     public void MergeTest(){
11         int[] actual = { 5, 1, 6, 2, 3, 4 };
12         int[] expected = { 1, 2, 3, 4, 5, 6 };
13         MergeSort.mergeSort(actual, actual.length);
14         assertEquals(expected, actual);
15     }
16
17 }
18

```



The screenshot shows an IDE with two tabs: 'test.java' and 'RadixSort.java'. The left sidebar displays test results for 'test.java', listing tests like HDT-3, scr, MergeTest, BubbleTest, GnomeTest, QuickTest, and RadixTest with their execution times. The main editor shows the Java code for these tests. The 'BubbleTest' method uses 'BubbleSort.bubbleSort' and 'assertArrayEquals2'. The 'GnomeTest' method uses 'GnomeSort.gnomeSort'. The 'QuickTest' method uses 'QuickSort.quickSort'. The 'RadixTest' method uses 'RadixSort.radixSort'. The code is written in Java and includes comments and annotations like '@Test'.

```
19
20
21 @Test
22 public void BubbleTest(){
23     ArrayList<Integer> actual = new ArrayList<>(Arrays.asList(...a: 5, 1, 6, 2, 3, 4));
24     int[] expected1 = { 1, 2, 3, 4, 5, 6 };
25     BubbleSort.bubbleSort(actual);
26     Integer a[] = actual.toArray(new Integer[actual.size()]);
27     assertArrayEquals2(expected1, a);
28 }
29
30 private void assertArrayEquals2(int[] expected1, Integer[] a) {
31 }
32
33 @Test
34 public void GnomeTest(){
35     int[] actual = { 5, 1, 6, 2, 3, 4 };
36     int[] expected = { 1, 2, 3, 4, 5, 6 };
37     GnomeSort.gnomeSort(actual, actual.length);
38     assertArrayEquals(expected, actual);
39 }
40
41 @Test
42 public void QuickTest(){
43     int[] actual = { 5, 1, 6, 2, 3, 4 };
44     int[] expected = { 1, 2, 3, 4, 5, 6 };
45     QuickSort.quickSort(actual, 0, actual.length - 1, new ComparadorEnteros<Integer>());
46     assertArrayEquals(expected, actual);
47 }
48
49 @Test
50 public void RadixTest(){
51     int[] actual = { 5, 1, 6, 2, 3, 4 };
52     int[] expected = { 1, 2, 3, 4, 5, 6 };
53     RadixSort.radixSort(actual);
54     assertArrayEquals(expected, actual);
55 }
56 }
```

VIDEO EXPLICATIVO DE MÉTODO UTILIZADO

Enlace del video de utilización del funcionamiento del algoritmo de elección Bubble Sort y Selection Sort.

Enlace: <https://youtu.be/O3QlFsg8j9Q>