Alumno—
Jesus Octavio Amarillas Amaya

ID—
207653

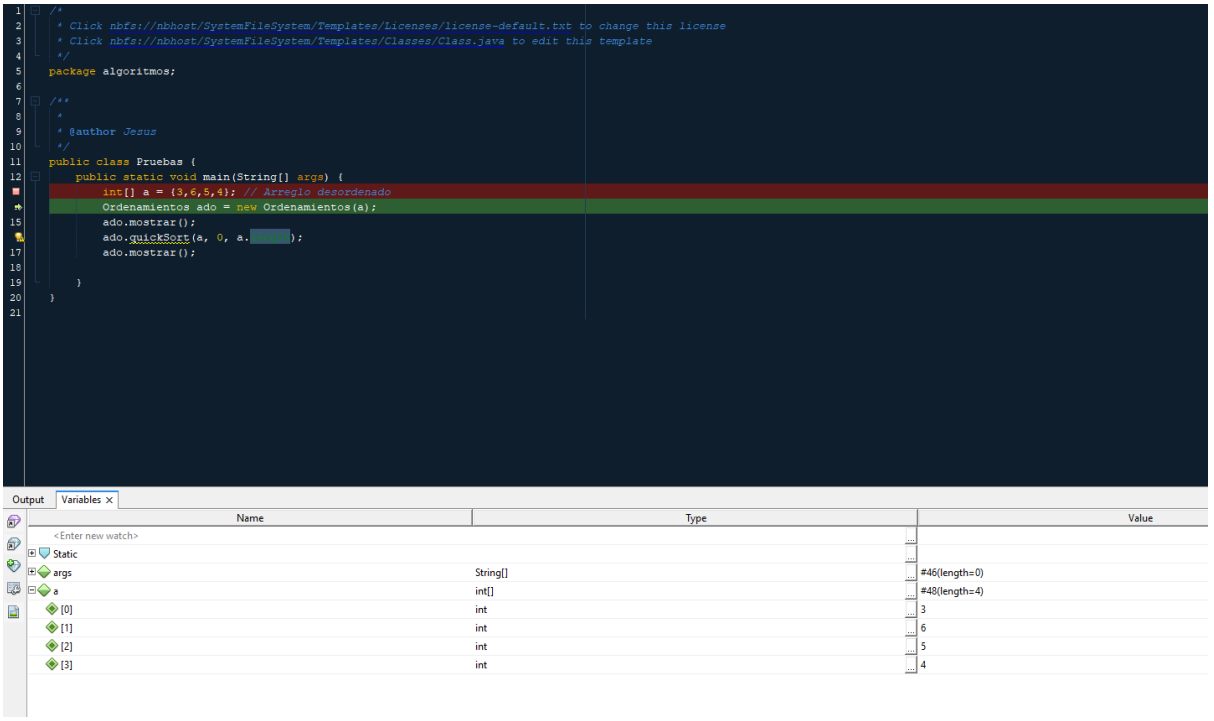
Asignación—
Algoritmo QuickSort

Materia—

Análisis de Algoritmos

Profesor—
Sergio Castellanos Bustamante

1. **Se declara el arreglo y se accede al método de quickSort con los argumentos que se mandan siendo el arreglo, el inicio de 0 y el final en la longitud del arreglo.**

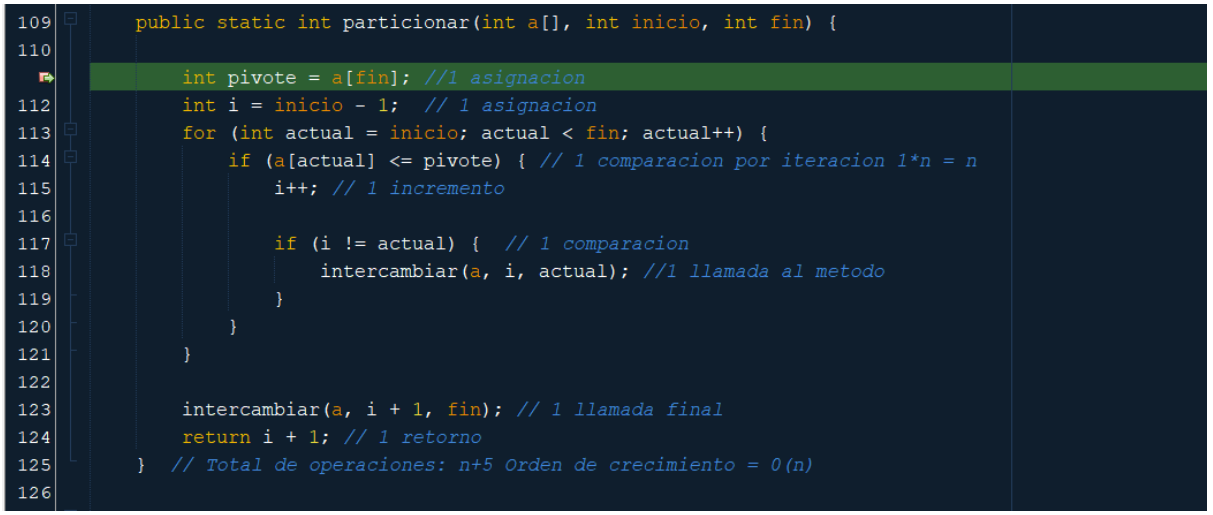


```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package algoritmos;

/**
 *
 * @author Jesus
 */
public class Pruebas {
    public static void main(String[] args) {
        int[] a = {3,6,5,4}; // Arreglo desordenado
        Ordenamientos ado = new Ordenamientos(a);
        ado.mostrar();
        ado.quickSort(a, 0, a.length);
        ado.mostrar();

    }
}
```

Output | Variables

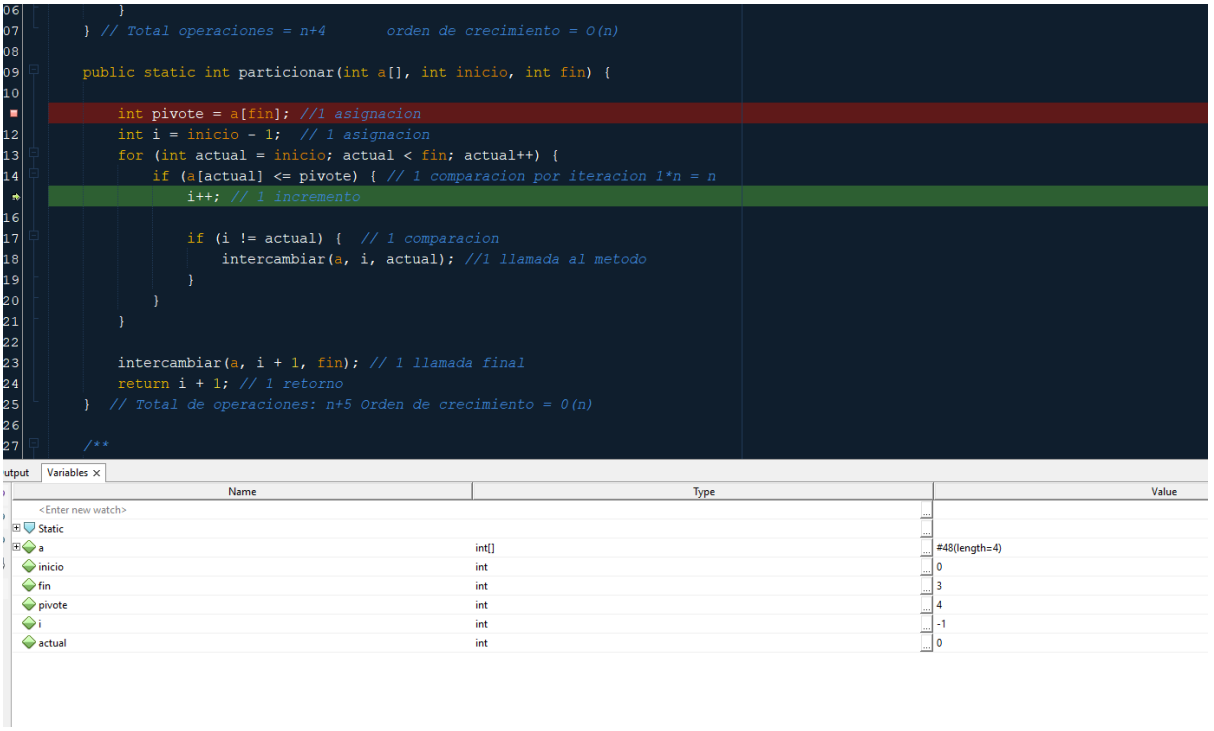
Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#46(length=0)
a	int[]	#48(length=4)
[0]	int	3
[1]	int	6
[2]	int	5
[3]	int	4

2. **Una vez accedido al método quicksort con los datos anteriormente mencionadas, se verifica que el valor de inicio sea menor que el del fin, si no se cumple se crea un pivote con un metodo auxiliar llamado particionar**



```java
    public static int particionar(int a[], int inicio, int fin) {

        int pivote = a[fin]; //1 asignacion
        int i = inicio - 1;  // 1 asignacion
        for (int actual = inicio; actual < fin; actual++) {
            if (a[actual] <= pivote) { // 1 comparacion por iteracion 1*n = n
                i++; // 1 incremento

                if (i != actual) {  // 1 comparacion
                    intercambiar(a, i, actual); //1 llamada al metodo
                }
            }
        }

        intercambiar(a, i + 1, fin); // 1 llamada final
        return i + 1; // 1 retorno
    }  // Total de operaciones: n+5 Orden de crecimiento = O(n)
```

3. **Para tener el valor de la variable anterior se recurre a un método auxiliar llamado “particionar” donde se mandan los mismos datos que el método QuickSort.**

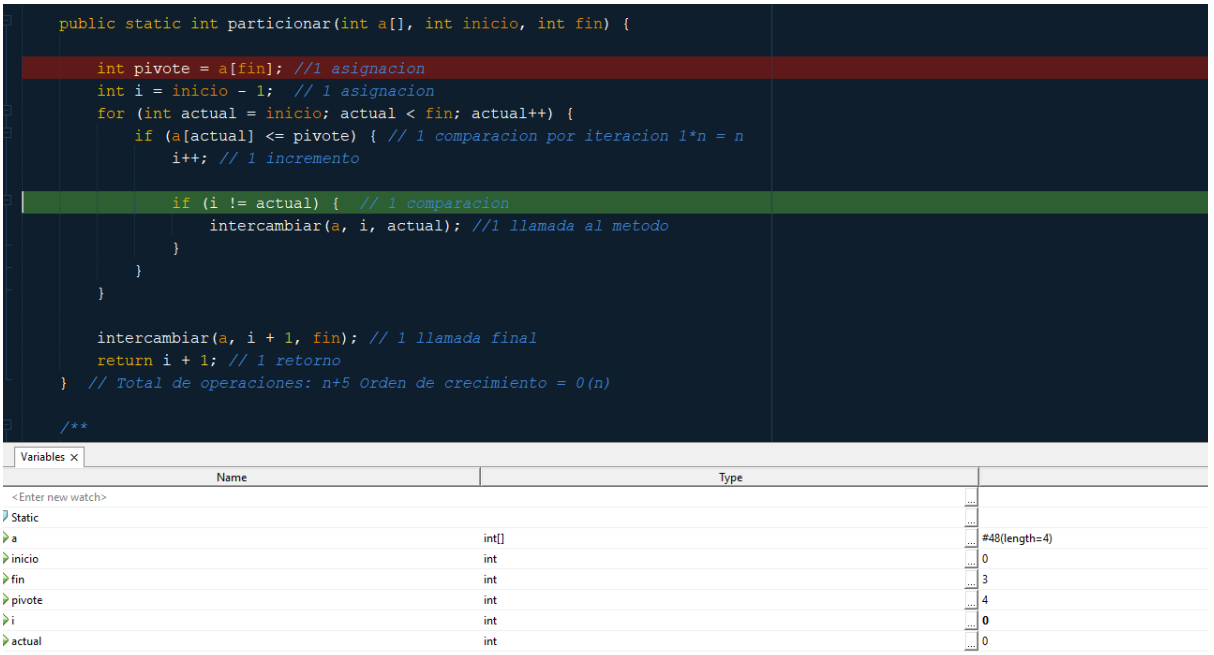


```
        }
    } // Total operaciones = n+4       orden de crecimiento = O(n)

    public static int particionar(int a[], int inicio, int fin) {

        int pivote = a[fin]; //1 asignacion
        int i = inicio - 1;  // 1 asignacion
        for (int actual = inicio; actual < fin; actual++) {
            if (a[actual] <= pivote) { // 1 comparacion por iteracion 1*n = n
                i++; // 1 incremento

                if (i != actual) {  // 1 comparacion
                    intercambiar(a, i, actual); //1 llamada al metodo
                }
            }
        }

        intercambiar(a, i + 1, fin); // 1 llamada final
        return i + 1; // 1 retorno
    }  // Total de operaciones: n+5 Orden de crecimiento = O(n)

    /**
```

utput | Variables ×

Name	Type	Value
<Enter new watch>		
Static		
a	int[]	#48(length=4)
inicio	int	0
fin	int	3
pivote	int	4
i	int	-1
actual	int	0

4. **Se crean 2 variables: ‘pivote’ con el valor del índice ‘fin’ del arreglo y ‘i’ con valor de ‘inicio’ menos 1.**

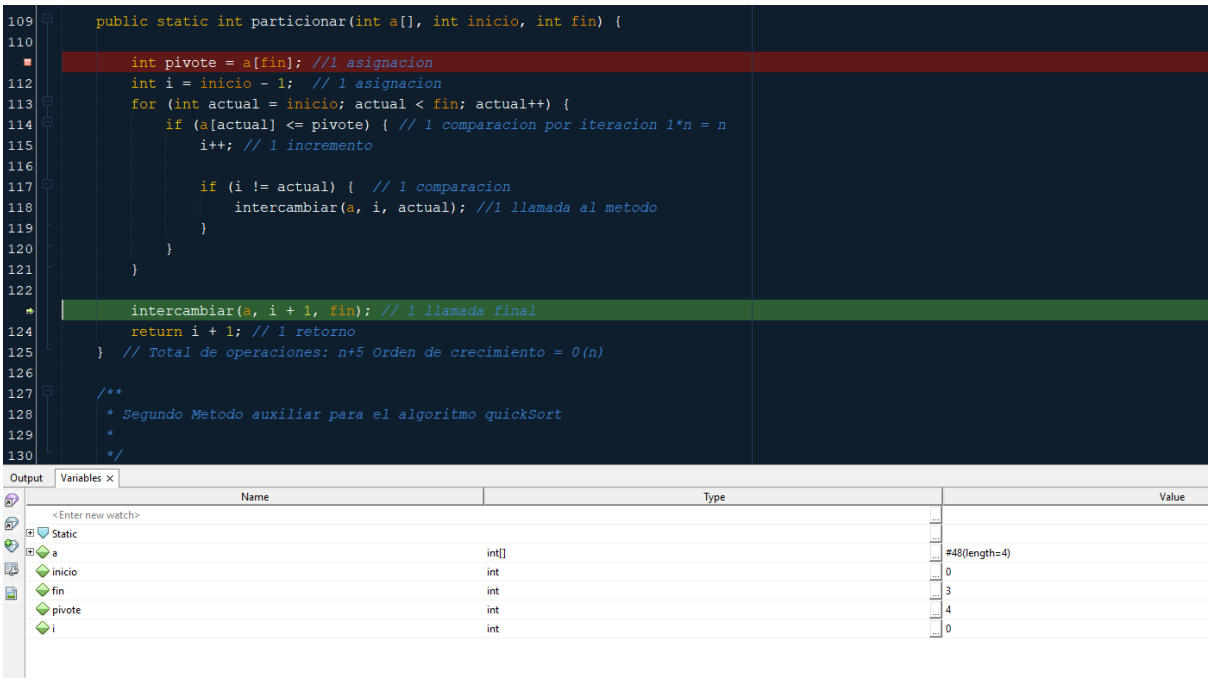


```
    public static int particionar(int a[], int inicio, int fin) {

        int pivote = a[fin]; //1 asignacion
        int i = inicio - 1;  // 1 asignacion
        for (int actual = inicio; actual < fin; actual++) {
            if (a[actual] <= pivote) { // 1 comparacion por iteracion 1*n = n
                i++; // 1 incremento

                if (i != actual) {  // 1 comparacion
                    intercambiar(a, i, actual); //1 llamada al metodo
                }
            }
        }

        intercambiar(a, i + 1, fin); // 1 llamada final
        return i + 1; // 1 retorno
    }  // Total de operaciones: n+5 Orden de crecimiento = O(n)

    /**
```

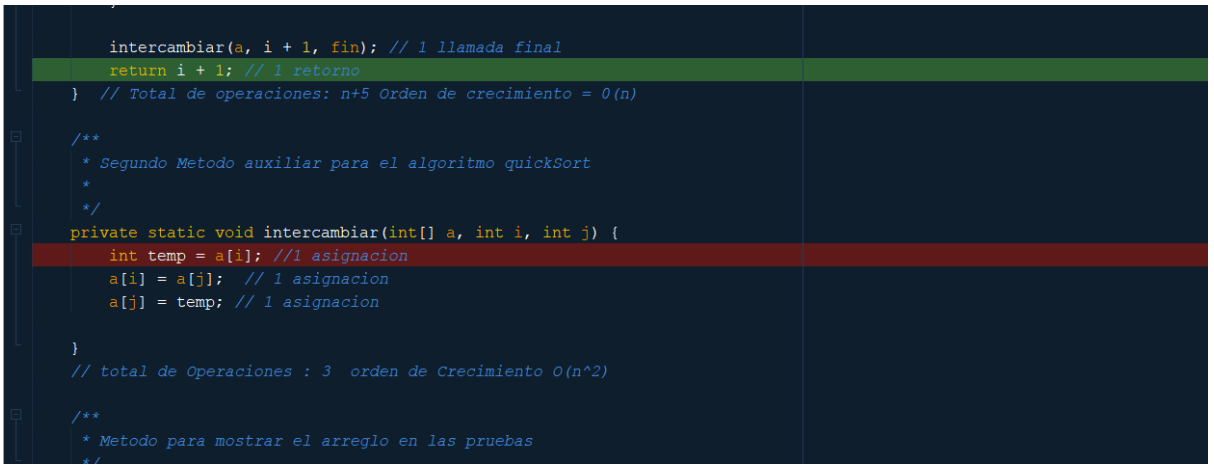
Variables ×

Name	Type	
<Enter new watch>		
Static		
a	int[]	#48(length=4)
inicio	int	0
fin	int	3
pivote	int	4
i	int	0
actual	int	0

5. Se utiliza un ciclo `for` donde se declara una variable llamada `actual`, que inicia con el valor de `inicio`. El ciclo continúa mientras `actual` sea menor que `fin`. Dentro del ciclo, hay una condición `if` que comprueba si el elemento en la posición `actual` del arreglo es menor o igual al valor del pivote. Si esta condición se cumple, se evalúa otra condición que verifica si `i` es diferente de `actual`; si también es verdadera, se llama a un método auxiliar llamado `intercambiar`.



6. El pivote se ubica en su posición definitiva utilizando el método `intercambiar`, y luego se retorna el valor de `i + 1` al método `quickSort`.



7. **Se aplica el método quickSort así mismo en la parte izquierda para los valores menores al pivote.**

```
97      public static void quickSort(int a[], int inicio, int fin) {
98
99          if (inicio < fin) { // n comparaciones
100
101             int pivote = particionar(a, inicio, fin); //1 llamada recursiva + 1 asignacion
102
103             quickSort(a, inicio, pivote - 1); //1 llamada recursiva
104
105             quickSort(a, pivote + 1, fin); //1 llamada recursiva
106         }
107     } // Total operaciones = n+4       orden de crecimiento = O(n)
108
109     public static int particionar(int a[], int inicio, int fin) {
110
111         int pivote = a[fin]; //1 asignacion
112         int i = inicio - 1;  // 1 asignacion
113         for (int actual = inicio; actual < fin; actual++) {
114             if (a[actual] <= pivote) { // 1 comparacion por iteracion 1*n = n
115                 i++; // 1 incremento
116
117                 if (i != actual) {  // 1 comparacion
118                     intercambiar(a, i, actual); //1 llamada al metodo
```

8. **Se aplica el método quickSort asi mismos ahora para los valores mayores al pivote.**

```
97      public static void quickSort(int a[], int inicio, int fin) {
98
99          if (inicio < fin) { // n comparaciones
100
101             int pivote = particionar(a, inicio, fin); //1 llamada recursiva + 1 asignacion
102
103             quickSort(a, inicio, pivote - 1); //1 llamada recursiva
104
105             quickSort(a, pivote + 1, fin); //1 llamada recursiva
106         }
107     } // Total operaciones = n+4       orden de crecimiento = O(n)
108
109     public static int particionar(int a[], int inicio, int fin) {
110
111         int pivote = a[fin]; //1 asignacion
112         int i = inicio - 1;  // 1 asignacion
113         for (int actual = inicio; actual < fin; actual++) {
114             if (a[actual] <= pivote) { // 1 comparacion por iteracion 1*n = n
115                 i++; // 1 incremento
116
117                 if (i != actual) {  // 1 comparacion
```

Output | Variables ×

Name	Type	Value
<Enter new watch>		
Static		
a	int[]	#48(length=4)
inicio	int	2
fin	int	3

9. **De nueva cuenta se vuelve a aplicar el metodo auxiliar “particionar” para seguir ordenando el arreglo**

```java
    public static int particionar(int a[], int inicio, int fin) {

        int pivote = a[fin]; //1 asignacion
        int i = inicio - 1;  // 1 asignacion
        for (int actual = inicio; actual < fin; actual++) {
            if (a[actual] <= pivote) { // 1 comparacion por iteracion 1*n = n
                i++; // 1 incremento

                if (i != actual) {  // 1 comparacion
                    intercambiar(a, i, actual); //1 llamada al metodo
                }
            }
        }

        intercambiar(a, i + 1, fin); // 1 llamada final
        return i + 1; // 1 retorno
    }  // Total de operaciones: n+5 Orden de crecimiento = O(n)

    /**
     * Segundo Metodo auxiliar para el algoritmo quickSort
```

10. Se repiten los pasos de crearse las variable pivote e i. y se vuelve a usar el método intercambiar para que se pueda intercambiar los valores del arreglo y despues regresar ‘i’ +1.

```java
        intercambiar(a, i + 1, fin); // 1 llamada final
        return i + 1; // 1 retorno
    }  // Total de operaciones: n+5 Orden de crecimiento = O(n)

    /**
     * Segundo Metodo auxiliar para el algoritmo quickSort
     *
     */
    private static void intercambiar(int[] a, int i, int j) {
        int temp = a[i]; //1 asignacion
        a[i] = a[j];  // 1 asignacion
        a[j] = temp; // 1 asignacion

    }
    // total de Operaciones : 3  orden de Crecimiento O(n^2)

    /**
     * Metodo para mostrar el arreglo en las pruebas
     */
    public void mostar() {
```

Variables

Name	Type	Value
<Enter new watch>		
Static		
a	int[]	#48(length=4)
inicio	int	2
fin	int	3
pivote	int	6
i	int	2

11. El metdod se vuelve a usar a si mismo para los demas valores mayores y menores al pivote

```
     * Metodo donde se aplica el algoritmo quicksort
     *
     * @param a
     * @param inicio
     * @param fin
     */
    public static void quickSort(int a[], int inicio, int fin) {

        if (inicio < fin) { // n comparaciones

            int pivote = particionar(a, inicio, fin); //1 llamada recursiva + 1 asignacion

            quickSort(a, inicio, pivote - 1); //1 llamada recursiva

            quickSort(a, pivote + 1, fin); //1 llamada recursiva
        }
    } // Total operaciones = n+4        orden de crecimiento = O(n)

    public static int particionar(int a[], int inicio, int fin) {

        int pivote = a[fin]; //1 asignacion
        int i = inicio - 1;  // 1 asignacion
```

Output | Variables

Name	Type	Value
<Enter new watch>		
Static		
a	int[]	#48(length=4)
inicio	int	2
fin	int	2

12. El método termina con el arreglo ordenado

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package algoritmos;

/**
 *
 * @author Jesus
 */
public class Pruebas {
    public static void main(String[] args) {
        int[] a = {3,6,5,4}; // Arreglo desordenado
        Ordenamientos ado = new Ordenamientos(a);
        ado.mostrar();
        ado.quickSort(a, 0, a.length-1);
        ado.mostrar();

    }
}
```

Output | Variables

Name	Type	Value
<Enter new watch>		
Static		
args	String[]	#46(length=0)
a	int[]	#48(length=4)
[0]	int	3
[1]	int	4
[2]	int	5
[3]	int	6
ado	Ordenamientos	#51