GUÍA DE LABORATORIO NO. 3

HANNA KATHERINE ABRIL GÓNGORA

JENNIFER NATALIA BELTRAN

FELIPE ARAUJO

UNIVERSIDAD MANUELA BELTRÁN

ESTRUCTURA DE DATOS

DOCENTE

HUGO ALFONSO ORTIZ BARRERO

BOGOTA DC 24 FEBRERO 2025

1. Trabajo a Realizar.

1.1.Algoritmos de ordenamiento.

1.1.1. Ordenamiento Burbuja (Bubble Sort)

El Bubble Sort es un algoritmo sencillo que compara elementos adyacentes y los intercambia si están en el orden incorrecto. Su complejidad es $O(n^2)$ en el peor caso.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:  # Intercambiar si el elemento es mayor que el siguiente
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

# Prueba del algoritmo
arr = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(arr))
```

1.1.2. Ordenamiento por Inserción (Insertion Sort)

El Insertion Sort funciona construyendo una lista ordenada de forma incremental, insertando elementos en la posición correcta. Es eficiente para listas pequeñas y su complejidad en el peor caso es $O(n^2)$.

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:  # Desplazar los elementos mayores
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Prueba del algoritmo
arr = [64, 34, 25, 12, 22, 11, 90]
print(insertion_sort(arr))
```

1.1.3. Ordenamiento por Selección (Selection Sort)

El Selection Sort selecciona el elemento más pequeño y lo intercambia con el primer elemento no ordenado. Tiene una complejidad de $O(n^2)$.

```python
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:  # Encontrar el mínimo en el resto del arreglo
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]  # Intercambiar
    return arr

# Prueba del algoritmo
arr = [64, 34, 25, 12, 22, 11, 90]
print(selection_sort(arr))
```

1.1.4. Ordenamiento Shell (Shell Sort)

El Shell Sort es una mejora del Insertion Sort que compara elementos separados por un "gap" y reduce este espacio hasta que se convierte en un Insertion Sort normal. Su complejidad varía según la secuencia de incrementos utilizada, pero en general es O(n log n).

```python
def shell_sort(arr):
    gap = len(arr) // 2
    while gap > 0:
        for i in range(gap, len(arr)):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr

# Prueba del algoritmo
arr = [64, 34, 25, 12, 22, 11, 90]
print(shell_sort(arr))
```

1.1.5. Ordenamiento Heap (Heap Sort

El Heap Sort utiliza una estructura de datos llamada montículo (heap) para ordenar los elementos. Se construye un heap máximo y se extrae el elemento mayor repetidamente. Su complejidad es O(n log n).

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

# Prueba del algoritmo
arr = [64, 34, 25, 12, 22, 11, 90]
print(heap_sort(arr))
```

1.1.6. Quick Sort

El Quick Sort utiliza un enfoque divide y vencerás, eligiendo un pivote, y luego ordena los elementos menores a la izquierda y los mayores a la derecha. Su complejidad promedio es O(n log n), aunque en el peor caso puede ser $O(n^2)$.

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # Se elige un pivote
    left = [x for x in arr if x < pivot]  # Menores al pivote
    middle = [x for x in arr if x == pivot]  # Igual al pivote
    right = [x for x in arr if x > pivot]  # Mayores al pivote
    return quick_sort(left) + middle + quick_sort(right)

# Prueba del algoritmo
arr = [64, 34, 25, 12, 22, 11, 90]
print(quick_sort(arr))
```

1.2.Pruebas de ordenado.

1.2.1. Tiempo calculado.

Cantidad Datos/ Algoritmo	1000	5000	10000	100000	1000000	10000000
Burbuja	19 ms | 18045900 ns	69 ms | 68645500 ns	158 ms | 158268200 ns	16038 ms | 16038275800 ns	1559991 ms | 1559990688200 ns	
Inserción	9 ms | 9269900 ns	27 ms | 26243000 ns	65 ms | 65023500 ns	3668 ms | 3667696100 ns	393233 ms | 393232934700 ns	
Selección	7 ms | 6410700 ns	8 ms | 8093100 ns	40 ms | 40013500 ns	2501 ms | 2502619400 ns	296862 ms | 296861635200 ns	
Shell	2 ms | 1665700 ns	9 ms | 8490700 ns	3 ms | 3681800 ns	42 ms | 42586700 ns	115 ms | 114978100 ns	311 ms | 311138900 ns
HeapSort	1 ms | 197200 ns	2 ms | 2773800 ns	3 ms | 3468500 ns	29 ms | 15394800 ns	114 ms | 114865100 ns	1105 ms | 1105588400 ns
Quick Sort	0 ms | 371900 ns	0 ms | 699400 ns	3 ms | 2909000 ns	26 ms | 25306500 ns	118 ms | 168217400 ns	953 ms | 952864300 ns

1.2.2. Pruebas de escritorio.

Arreglo de entrada: [29,10,14,37,13,12,19,42,9,27]

- **Bubble Sort.**

Bubble Sort compara y cambia de posición los elementos adyacentes si están en el orden incorrecto. Se repite este proceso hasta que el arreglo esté completamente ordenado. En cada iteración, el número más grande "burbujea" hasta su posición final.

Iteración	Estado del arreglo	Intercambios realizados
1	**10 14 29** 13 12 19 37 9 27 42	29↔14, 29↔13, 29↔12, 29↔19, 29↔9, 29↔27
2	**10 14 13** 12 19 29 9 27 37 42	14↔13, 14↔12, 29↔9, 29↔27
3	**10 13 12** 14 19 9 27 29 37 42	13↔12, 19↔9
4	**10 12 13** 14 9 19 27 29 37 42	14↔9
5	**10 12 13** 9 14 19 27 29 37 42	13↔9
6	**10 12 9** 13 14 19 27 29 37 42	12↔9
7	**10 9 12** 13 14 19 27 29 37 42	10↔9
8	**9 10 12 13 14 19 27 29 37 42**	Ninguno

- **Insertion Sort.**

Insertion Sort toma cada elemento y lo coloca en la posición correcta dentro de la parte ordenada del arreglo. Se comparan los valores de derecha a izquierda hasta encontrar la posición adecuada.

Iteración	Estado del arreglo	Intercambios realizados
1	**9** 10 14 37 13 12 19 42 29 27	9
2	9 **10** 14 37 13 12 19 42 29 27	10

3	9 10 **12** 37 13 14 19 42 29 27	12
4	9 10 12 **13** 37 14 19 42 29 27	13
5	9 10 12 13 **14** 37 19 42 29 27	14
6	9 10 12 13 14 **19** 37 42 29 27	19
7	9 10 12 13 14 19 **27** 42 29 37	27
8	9 10 12 13 14 19 27 **29** 42 37	29
9	9 10 12 13 14 19 27 29 **37** 42	37

- **Quick Sort.**

Quick Sort selecciona un pivote y divide el arreglo en dos partes: menores y mayores que el pivote. Luego, repite el proceso recursivamente en cada subarreglo.

Iteración	Estado del arreglo	Intercambios realizados
1	**10 9** 14 37 13 12 19 42 29 27	27
2	**9 10 12 14** 13 19 27 29 37 42	19
3	**9 10 12 13 14** 19 27 29 37 42	14
4	**9 10 12 13 14** 19 27 29 37 42	Ordenado

- **Shell Sort**

Shell Sort es una mejora de Insertion Sort. Divide el arreglo en grupos usando un "gap" o intervalo y ordena los elementos en cada grupo. Luego, el **gap** se reduce hasta llegar a 1, momento en el que se aplica **Insertion Sort**.

Iteración	Estado del arreglo	Intercambios realizados
1	**29 10 14 37 13 12 19 42 9 27**	5
2	**12 10 14 37 13 29 19 42 9 27**	5

3	**12 9 14 37 13 29 19 42 10 27**	5
4	**9 10 14 37 13 29 12 42 19 27**	3
5	**9 10 12 27 13 29 14 42 19 37**	1
6	**9 10 12 13 14 19 27 29 37 42**	1

1. Se inicia con un **gap** de 5 y se ordenan los elementos en esos intervalos.
2. Luego, el **gap** se reduce a 3 y nuevamente se ordenan los elementos.
3. Finalmente, con un **gap** de 1, el algoritmo se convierte en **Insertion Sort**, dejando el arreglo ordenado.

- **Heap Sort.**

Insertion Sort toma cada elemento y lo coloca en la posición correcta dentro de la parte ordenada del arreglo. Se comparan los valores de derecha a izquierda hasta encontrar la posición adecuada.

Iteración	**Estado del arreglo**	**Intercambios realizados**
1	**42 29 19 37 27 12 14 10 9 13**	Construcción del heap
2	**13 29 19 37 27 12 14 10 9 42**	Intercambio 42↔13
3	**37 29 19 13 27 12 14 10 9 42**	Reajuste del heap
4	**9 29 19 13 27 12 14 10 37 42**	Intercambio 37↔9
5	**29 27 19 13 9 12 14 10 37 42**	Reajuste del heap
6	**10 27 19 13 9 12 14 29 37 42**	Intercambio 29↔10
7	**27 14 19 13 9 12 10 29 37 42**	Reajuste del heap
8	**10 14 19 13 9 12 27 29 37 42**	Intercambio 27↔10
9	**9 12 14 13 10 19 27 29 37 42**	Intercambio 19↔9
10	**9 10 12 13 14 19 27 29 37 42**	Arreglo ordenado

1. Se construye un **heap máximo** donde el mayor elemento está en la raíz.

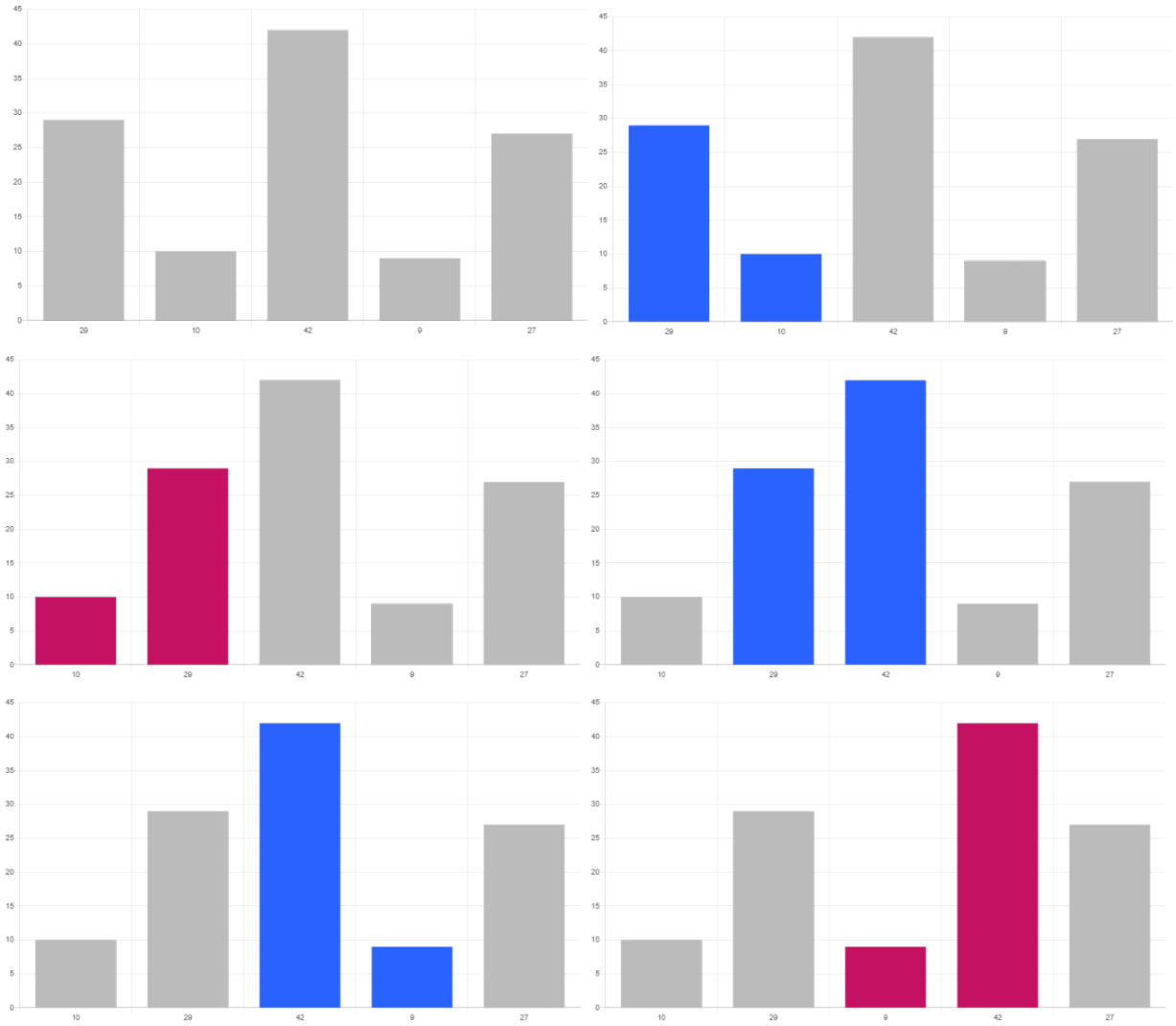
2. Se intercambia el primer y el último elemento, colocando el mayor en su posición final.

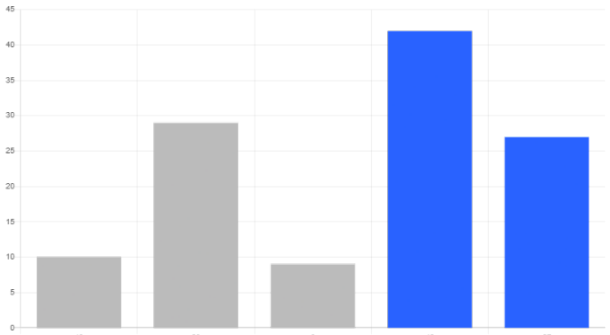
3. Se reajusta el heap y se repite el proceso hasta que el arreglo esté completamente ordenado.

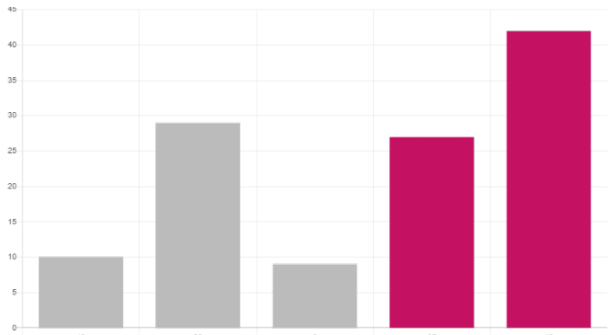
1.3.Explicación grafica de los algoritmos.

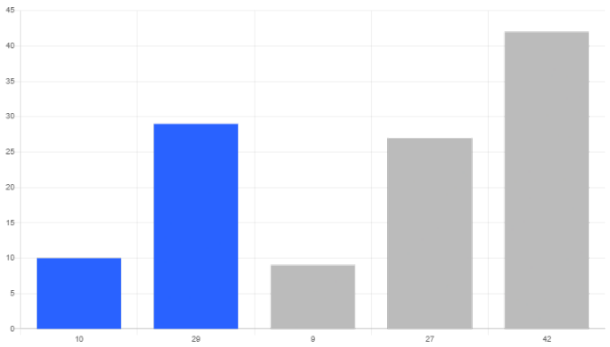
Arreglo de entrada: [29, 10,42, 9, 27]

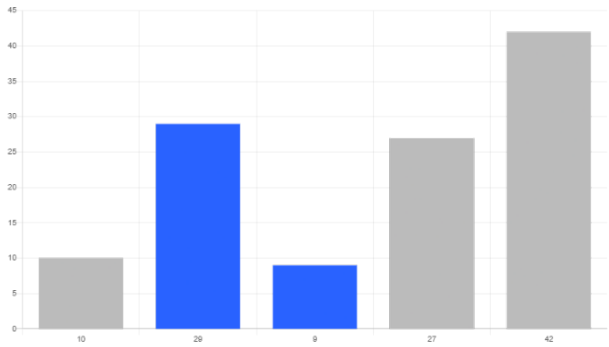
1.3.1. Bubble Sort.

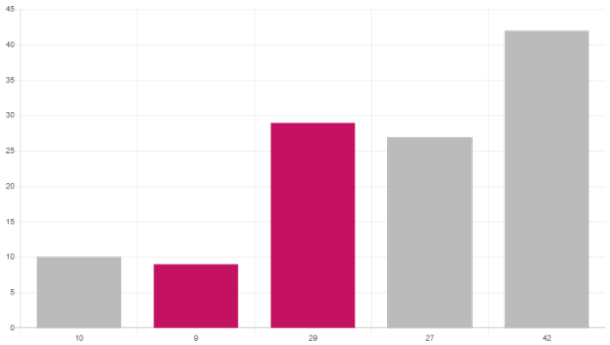


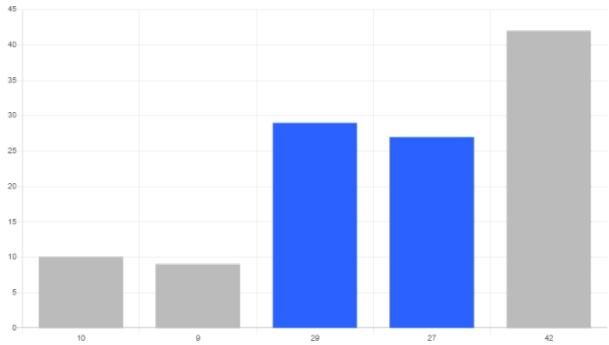

45
40
35
30
25
20
15
10
5
0
10
29
9
42
27

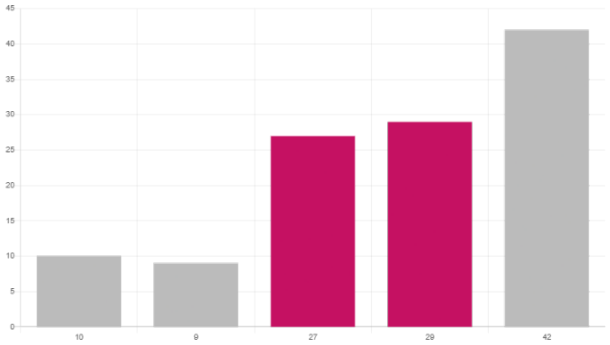

45
40
35
30
25
20
15
10
5
0
10
29
9
27
42

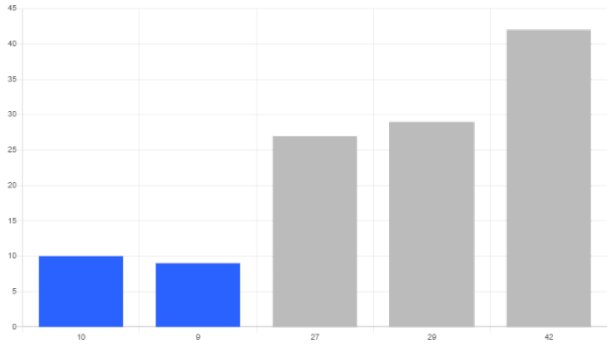

45
40
35
30
25
20
15
10
5
0
10
29
9
27
42

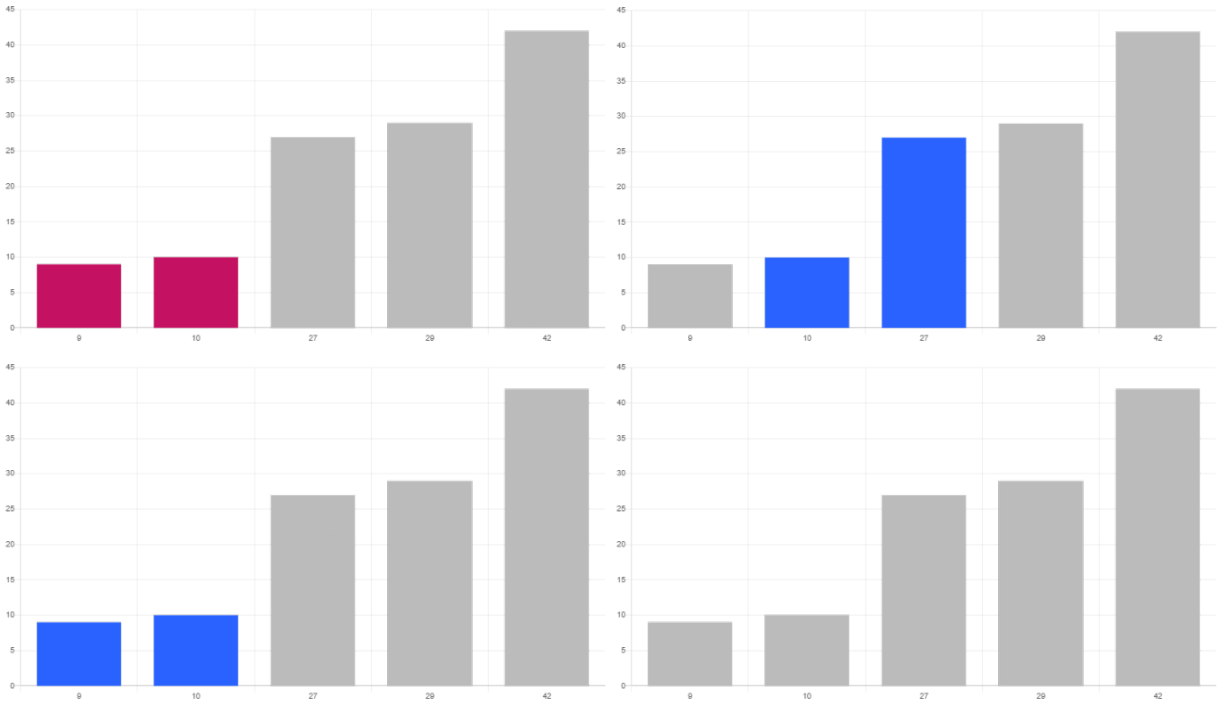

45
40
35
30
25
20
15
10
5
0
10
29
9
27
42


45
40
35
30
25
20
15
10
5
0
10
9
29
27
42


45
40
35
30
25
20
15
10
5
0
10
9
29
27
42


45
40
35
30
25
20
15
10
5
0
10
9
27
29
42


45
40
35
30
25
20
15
10
5
0
10
9
27
29
42

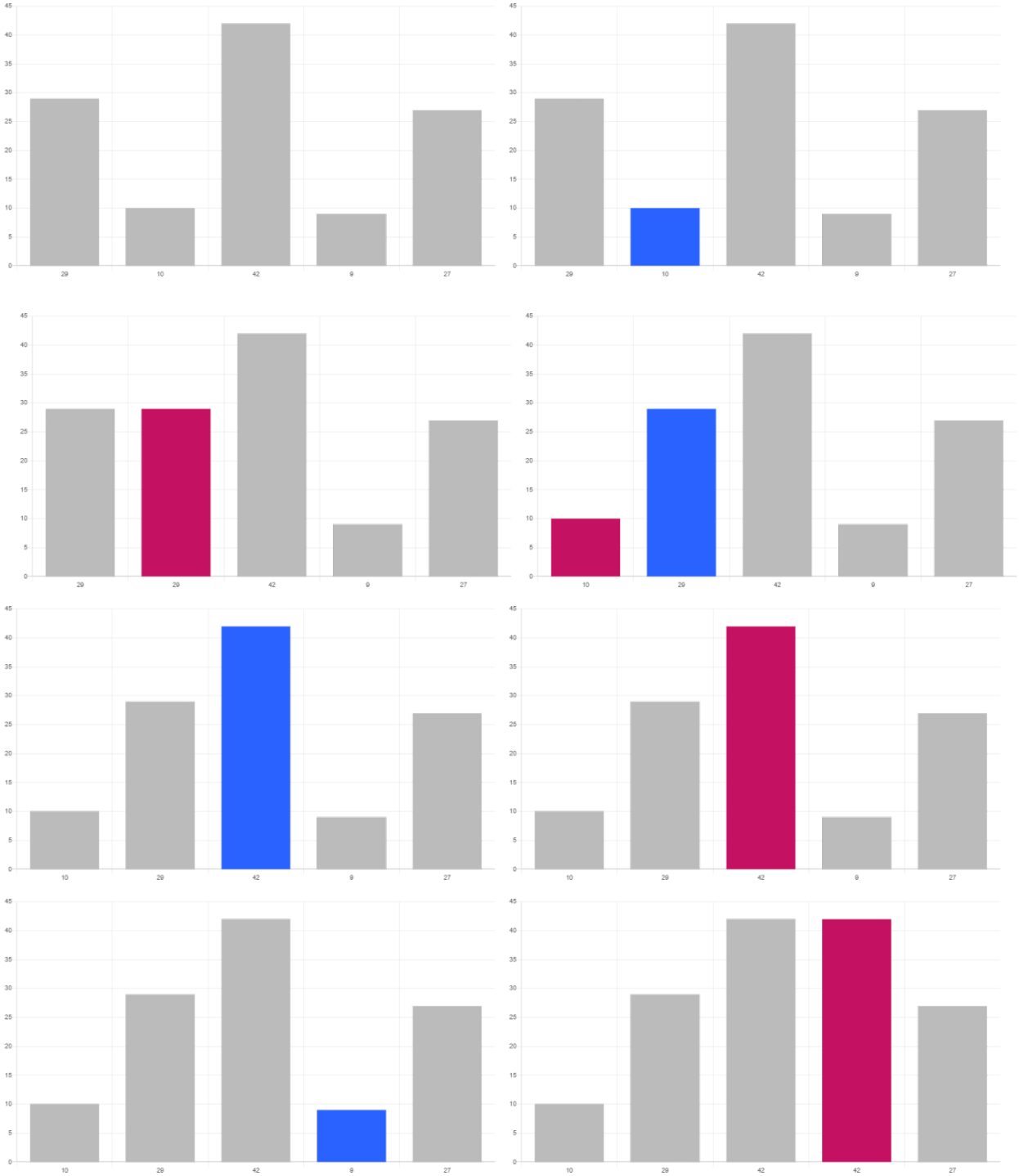


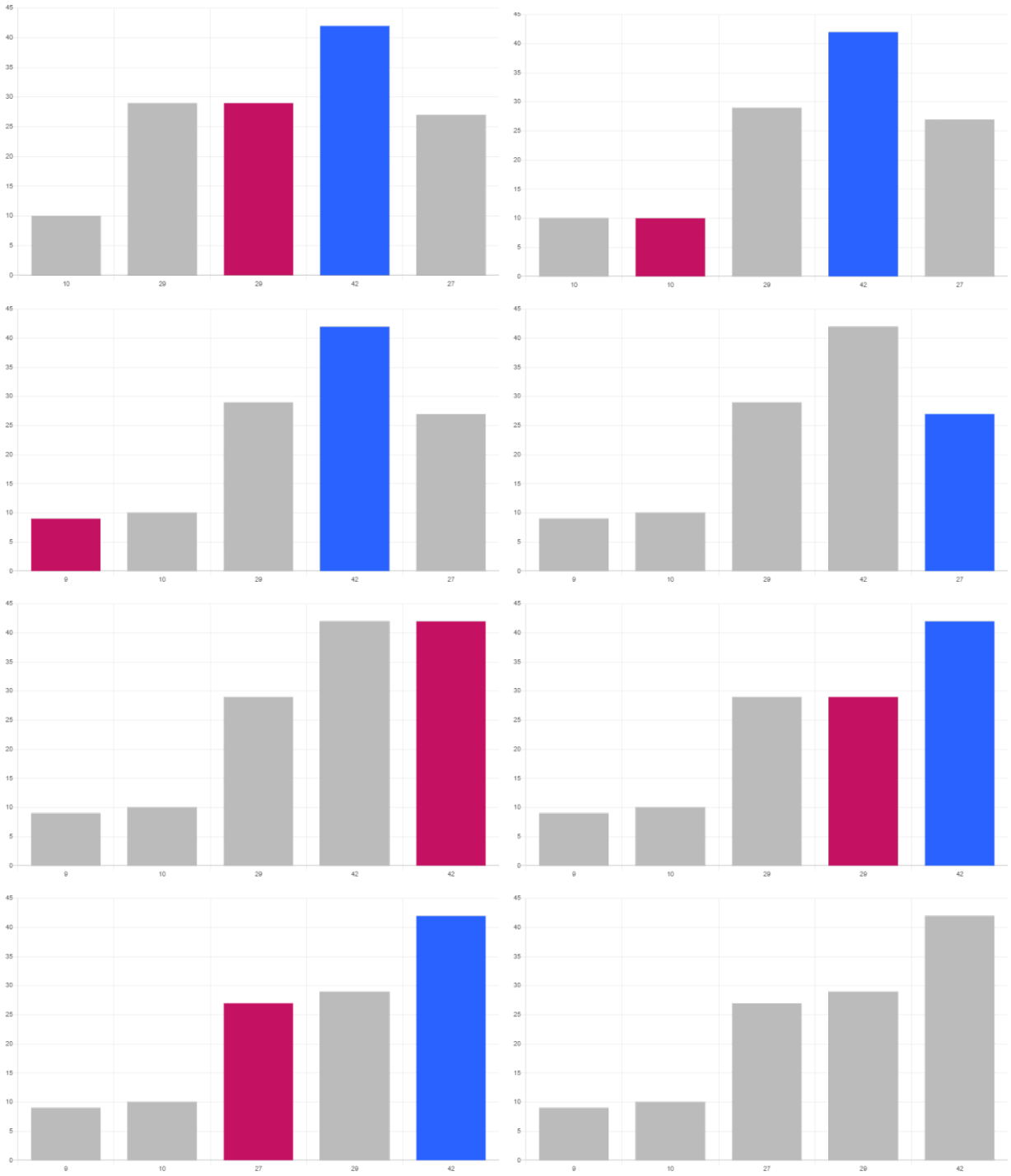
Link para revisar el Sort animado: https://algorithm-visualizer.org/brute-force/bubble-sort

Codigo Ejecutado:

```java
import org.algorithm_visualizer.*;

import java.util.Arrays;

public class Main {

    private static ChartTracer chartTracer = new ChartTracer();
    private static LogTracer logTracer = new LogTracer("Console");

    // Arreglo de prueba fijo
    private static Integer[] array = {29, 10,42, 9, 27};

    public static void main(String[] args) {

        int length = array.length;

        logTracer.printf("Original array = %s\n", Arrays.toString(array));
        chartTracer.set(array);
        Layout.setRoot(new VerticalLayout(new Commander[]{chartTracer, logTracer}));
        Tracer.delay();

        boolean flag;

        for (int i = length - 1; i > 0; i--) {
            flag = true;
            for (int j = 0; j < i; j++) {
                chartTracer.select(j);
                chartTracer.select(j + 1);
                Tracer.delay();
                if (array[j] > array[j + 1]) {
                    logTracer.printf("Swap %s and %s\n", array[j], array[j + 1]);
                    swap(j, j + 1, array);
                    flag = false;
                }
                chartTracer.deselect(j);
                chartTracer.deselect(j + 1);
            }
            if (flag) {
                break;
            }
        }

        logTracer.printf("Sorted array = %s\n", Arrays.toString(array));
    }

    private static void swap(int x, int y, Integer[] array) {
        int temp = array[x];
        array[x] = array[y];
        array[y] = temp;
        chartTracer.patch(x, array[x]);
        chartTracer.patch(y, array[y]);
        Tracer.delay();
        chartTracer.depatch(x);
        chartTracer.depatch(y);
    }
}

```

1.3.2. Insertion Sort.



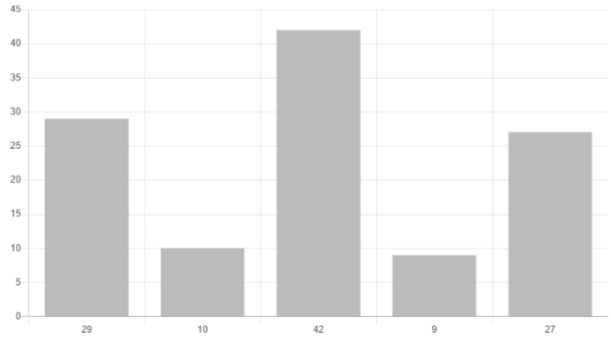


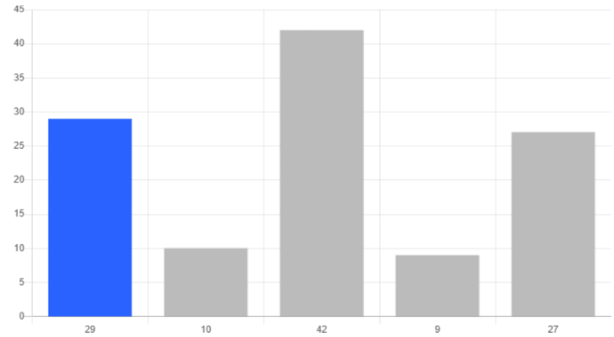
Link para revisar el Sort animado: https://algorithm-visualizer.org/brute-force/insertion-sort

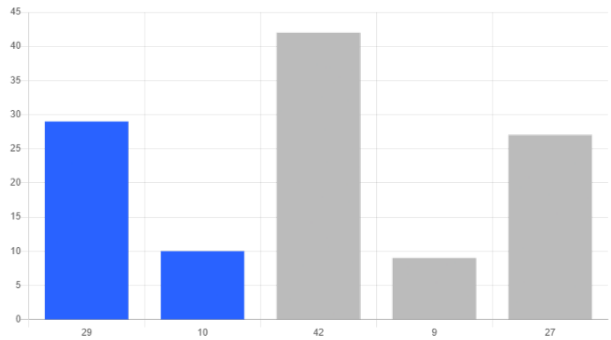
Codigo Ejecutado:

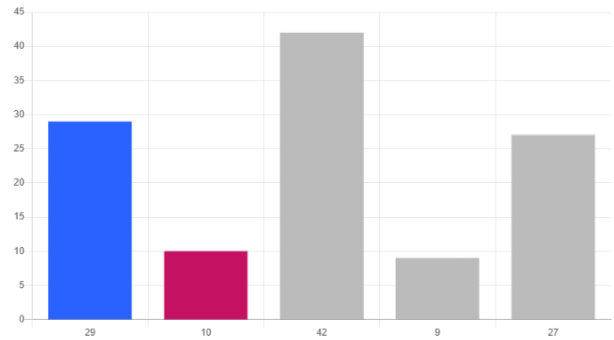
```java
import org.algorithm_visualizer.*;
import java.util.Arrays;

class Main {

    private static ChartTracer chartTracer = new ChartTracer();
    private static LogTracer logTracer = new LogTracer("Console");

    // Arreglo de prueba fijo
    private static Integer[] array = {29, 10, 42, 9, 27};

    public static void main(String[] args) {
        int length = array.length;
        Layout.setRoot(new VerticalLayout(new Commander[]{chartTracer, logTracer}));
        logTracer.printf("Original array = %s\n", Arrays.toString(array));
        chartTracer.set(array);
        Tracer.delay();

        for (int i = 1; i < length; i++) {
            int j = i - 1;
            int temp = array[i];
            chartTracer.select(i);
            Tracer.delay();
            logTracer.printf("Insert %s\n", temp);

            while (j >= 0 && array[j] > temp) {
                array[j + 1] = array[j];
                chartTracer.patch(j + 1, array[j + 1]);
                Tracer.delay();
                chartTracer.depatch(j + 1);
                j--;
            }

            array[j + 1] = temp;
            chartTracer.patch(j + 1, temp);
            Tracer.delay();
            chartTracer.depatch(j + 1);
            chartTracer.deselect(i);
        }

        logTracer.printf("Sorted array = %s\n", Arrays.toString(array));
    }
}

```

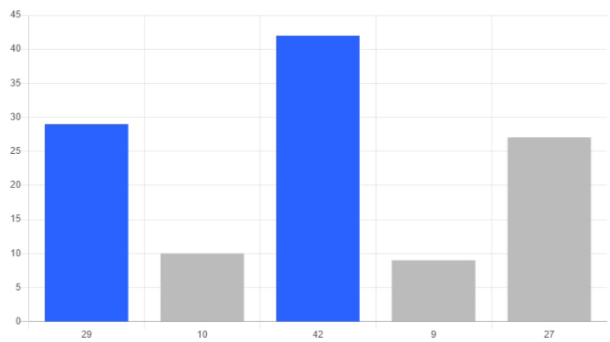
1.3.3. Selection Sort.

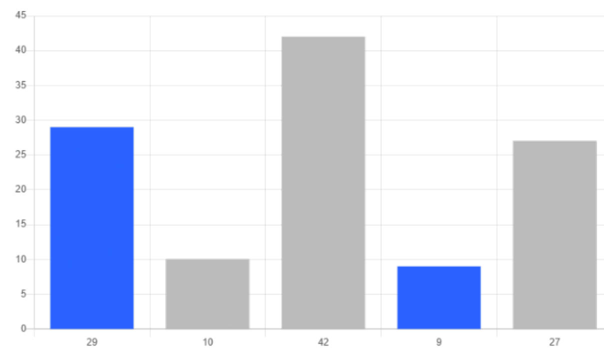

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

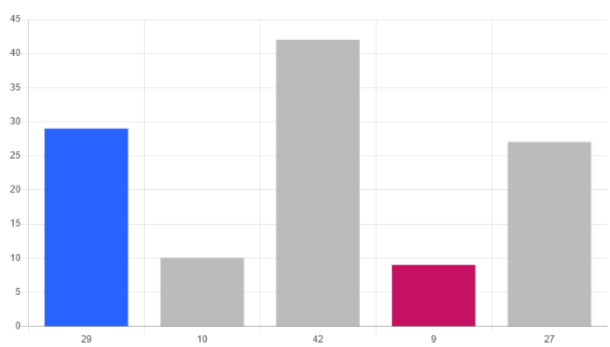

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

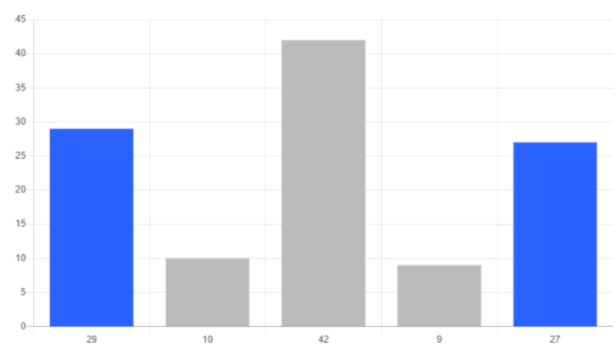

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

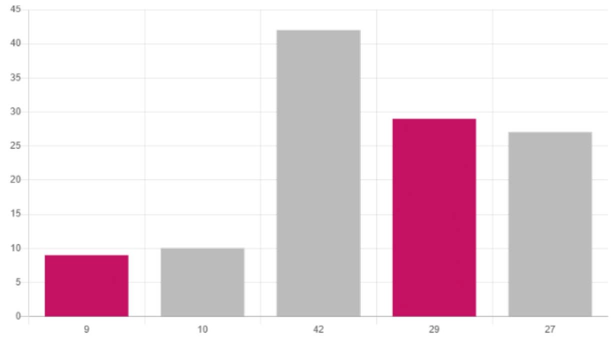

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

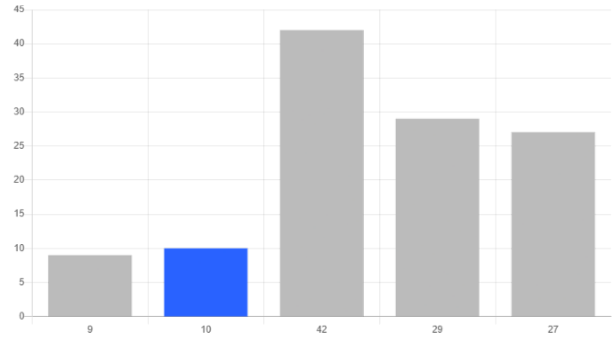

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

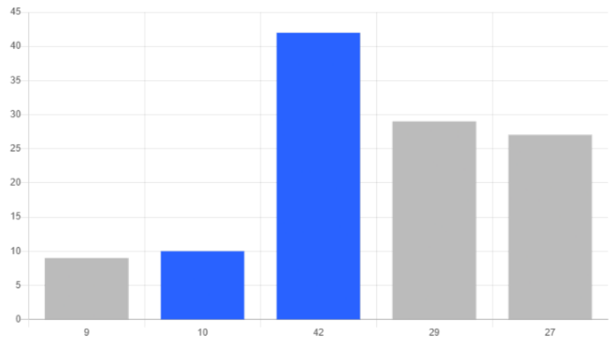

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

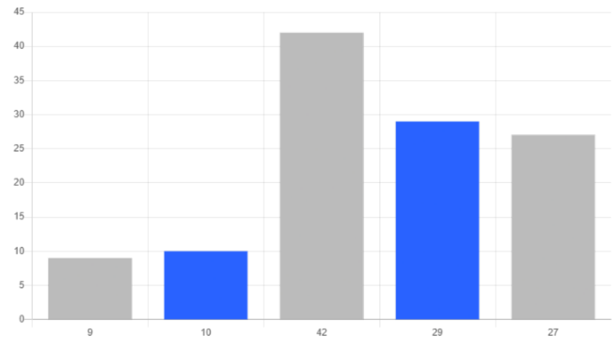

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

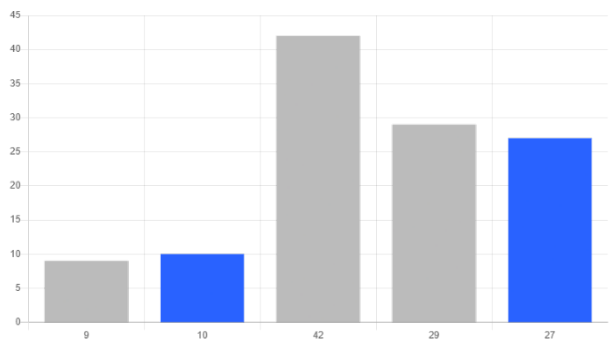

45
40
35
30
25
20
15
10
5
0
29
10
42
9
27

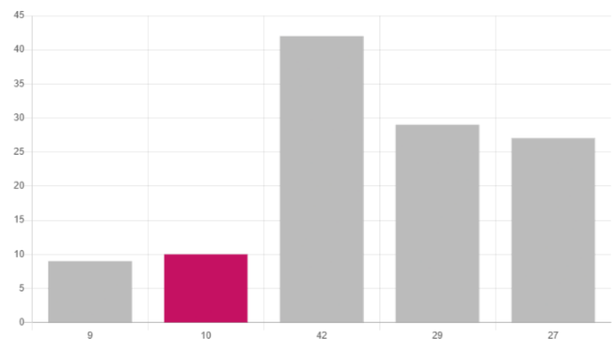

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

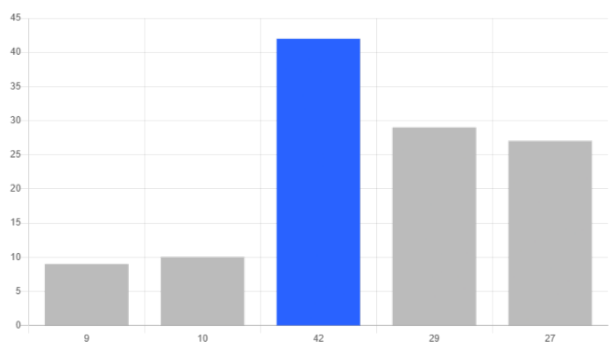

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

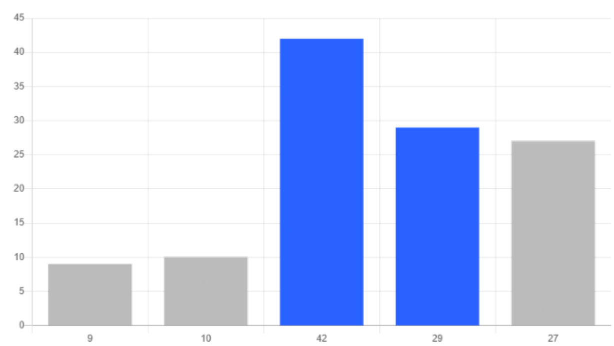

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

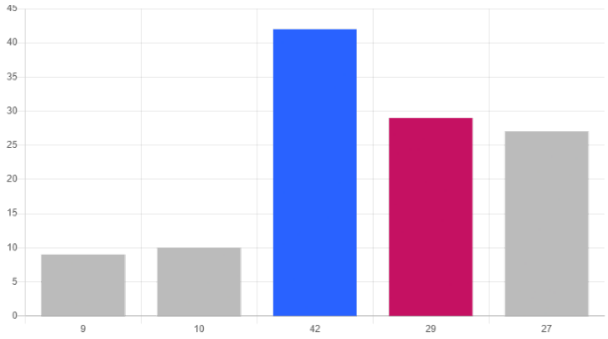

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

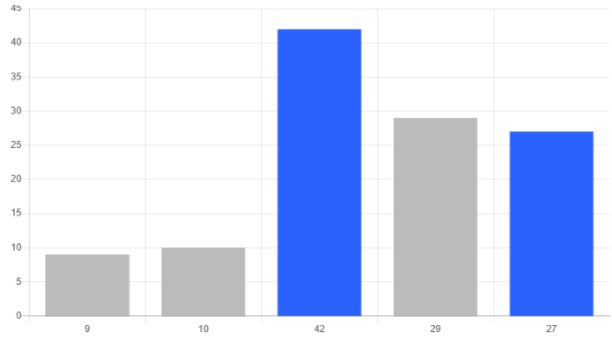

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

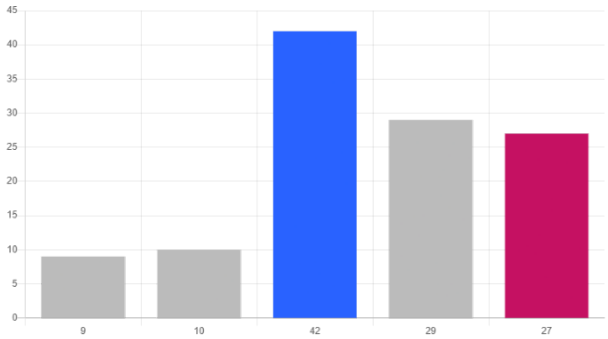

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

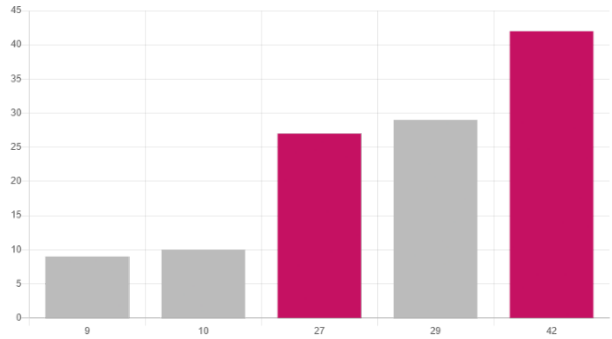

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

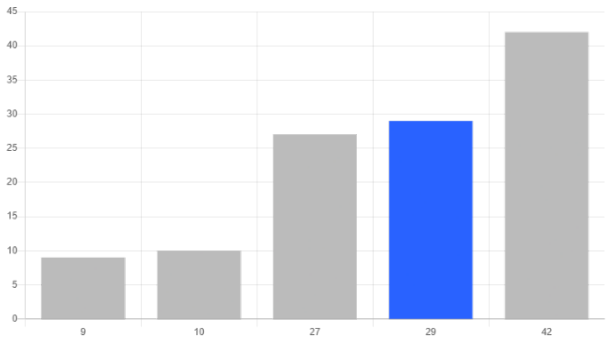

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

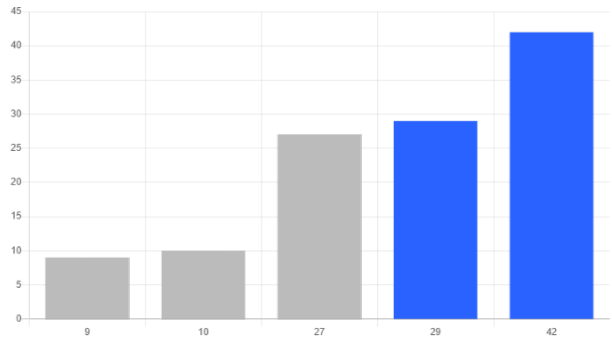

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27

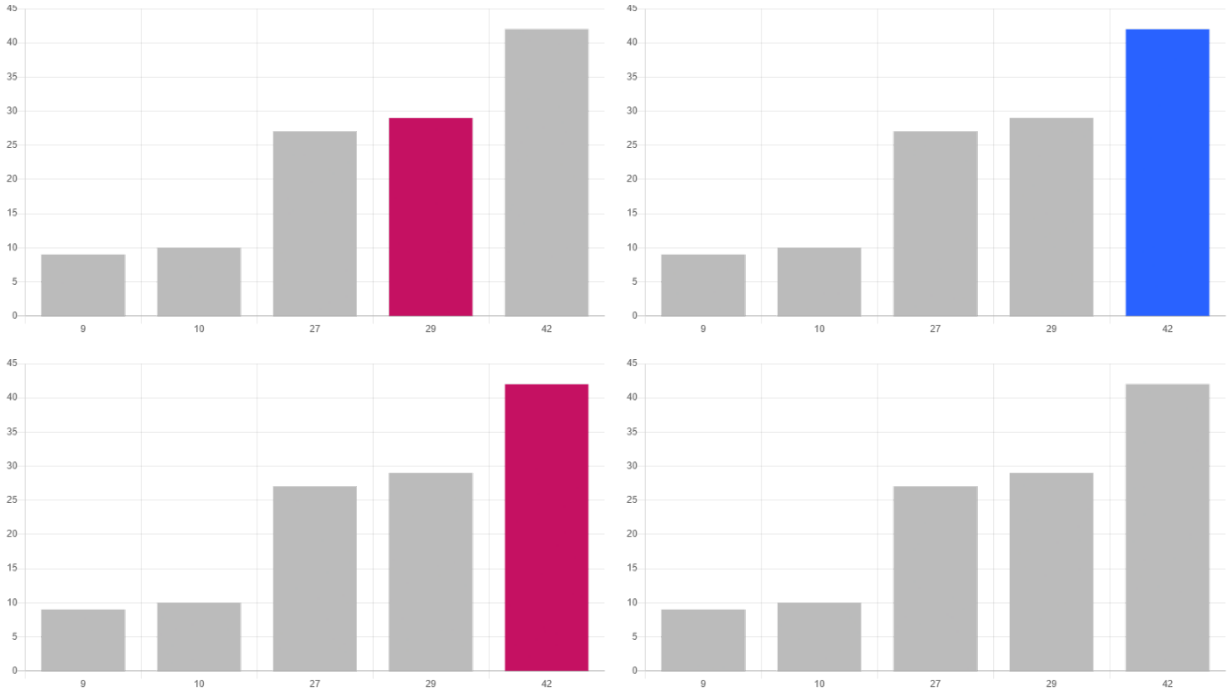

45
40
35
30
25
20
15
10
5
0
9
10
42
29
27


45
40
35
30
25
20
15
10
5
0
9
10
42
29
27


45
40
35
30
25
20
15
10
5
0
9
10
27
29
42


45
40
35
30
25
20
15
10
5
0
9
10
27
29
42


45
40
35
30
25
20
15
10
5
0
9
10
27
29
42

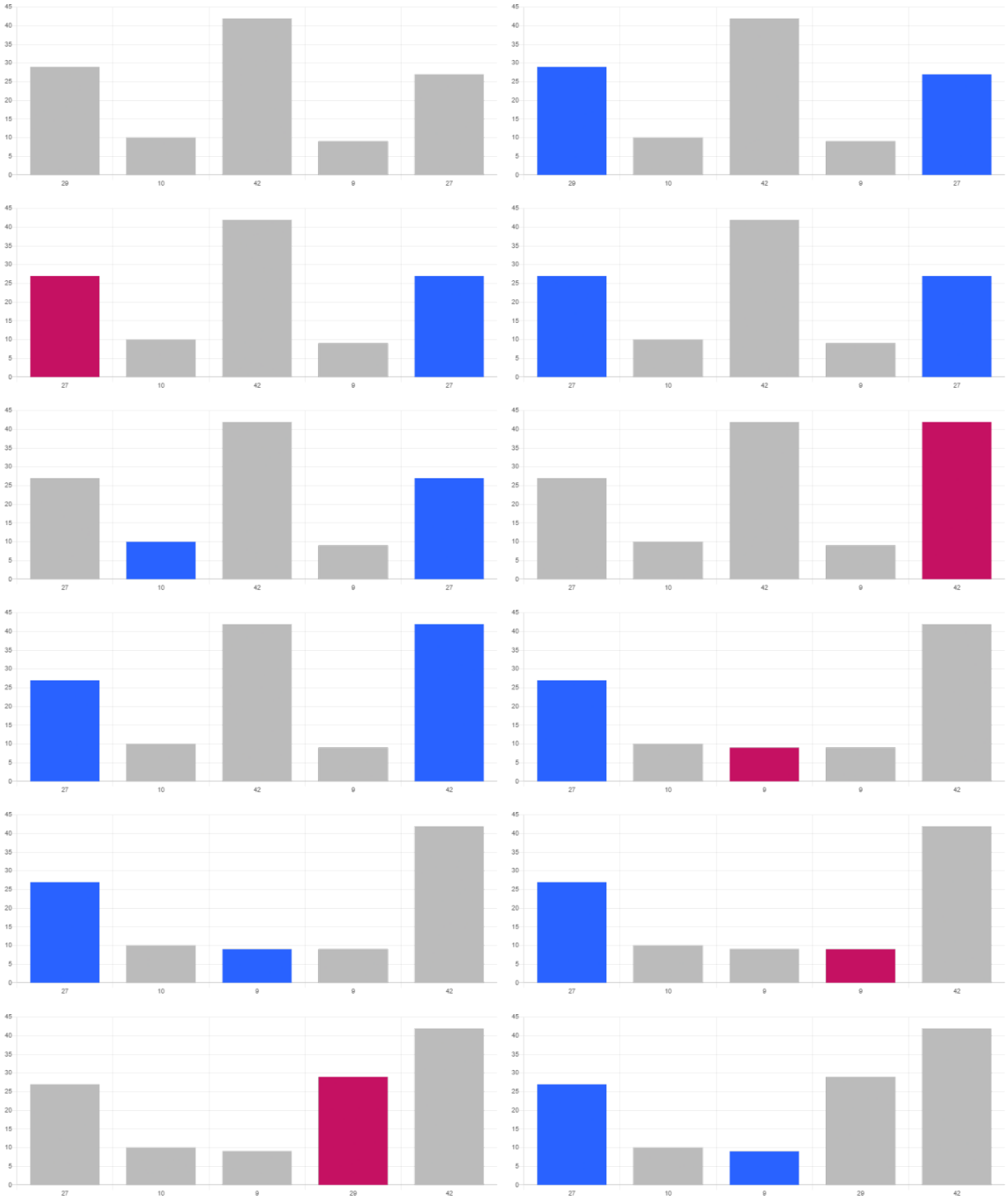


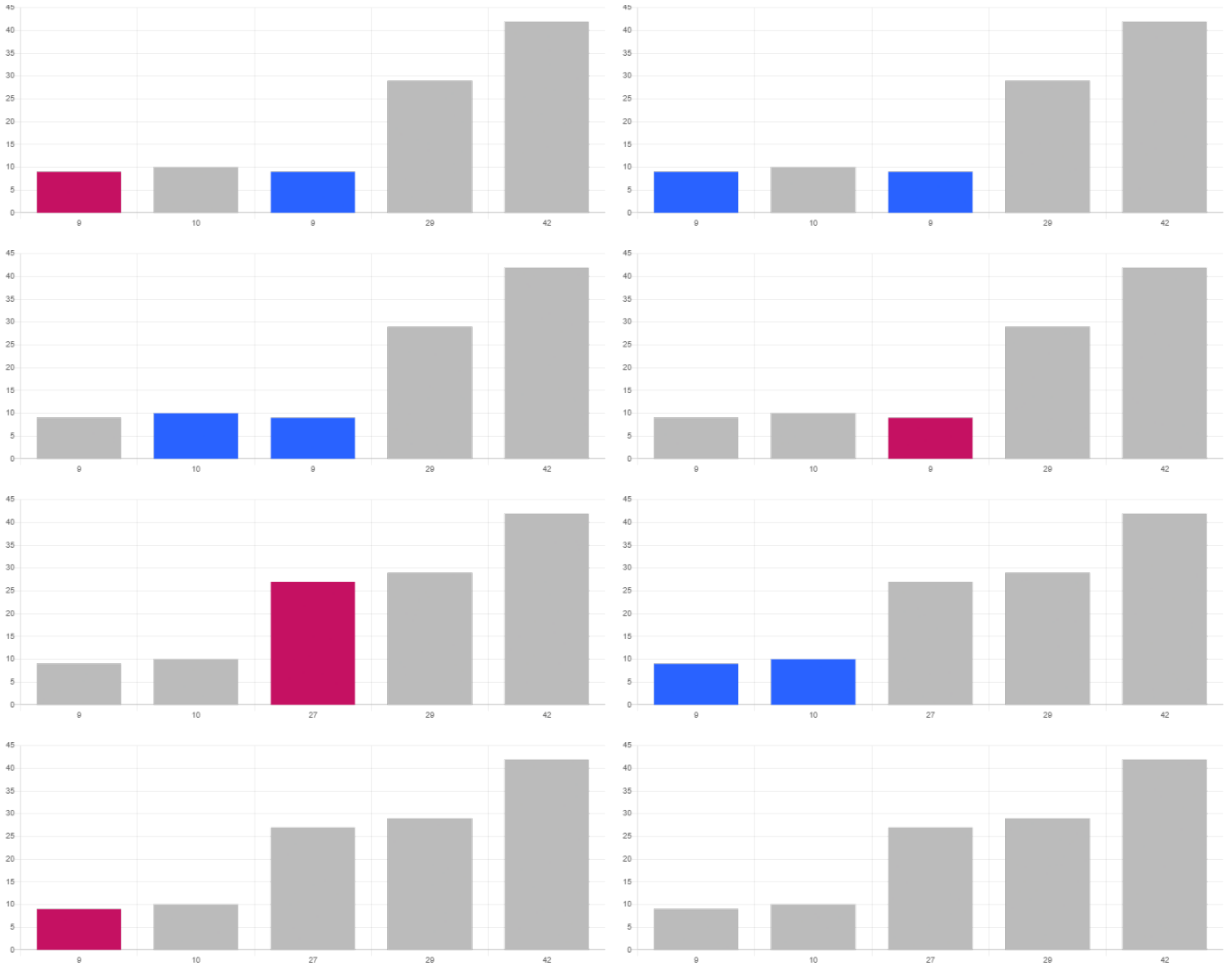
Link para revisar el Sort animado: https://algorithm-visualizer.org/brute-force/selection-sort

Codigo Ejecutado:

```java
import org.algorithm_visualizer.*;

import java.util.Arrays;

public class Main {

    private static ChartTracer chartTracer = new ChartTracer();
    private static LogTracer logTracer = new LogTracer("Console");

    // Arreglo fijo
    private static Integer[] array = {29, 10, 42, 9, 27};

    public static void main(String[] args) {
        int length = array.length;
        Layout.setRoot(new VerticalLayout(new Commander[]{chartTracer, logTracer}));
        logTracer.printf("Original array = %s\n", Arrays.toString(array));
        chartTracer.set(array);
        Tracer.delay();

        int minIndex;

        for (int i = 0; i < length; i++) {
            chartTracer.select(i);
            Tracer.delay();
            minIndex = i;

            for (int j = i + 1; j < length; j++) {
                chartTracer.select(j);
                Tracer.delay();

                if (array[j] < array[minIndex]) {
                    chartTracer.patch(j, array[j]);
                    Tracer.delay();
                    chartTracer.depatch(j);
                    minIndex = j;
                }

                chartTracer.deselect(j);
            }

            swap(minIndex, i, array);
            chartTracer.deselect(i);
        }

        logTracer.printf("Sorted array = %s\n", Arrays.toString(array));
    }

    private static void swap(int x, int y, Integer[] array) {
        int temp = array[x];
        array[x] = array[y];
        array[y] = temp;
        chartTracer.patch(x, array[x]);
        chartTracer.patch(y, array[y]);
        Tracer.delay();
        chartTracer.depatch(x);
        chartTracer.depatch(y);
    }
}

```

1.3.4. Quick Sort.



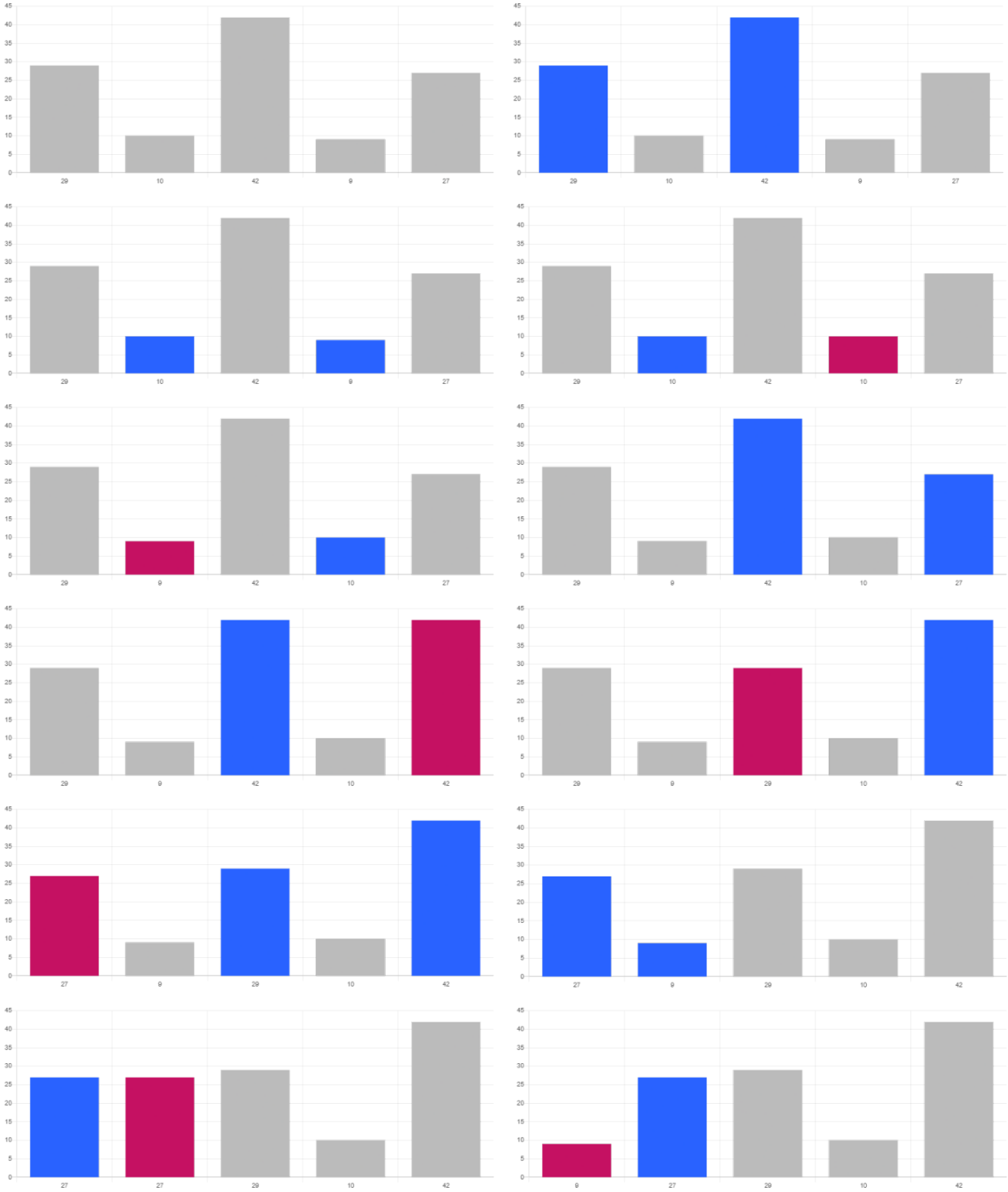


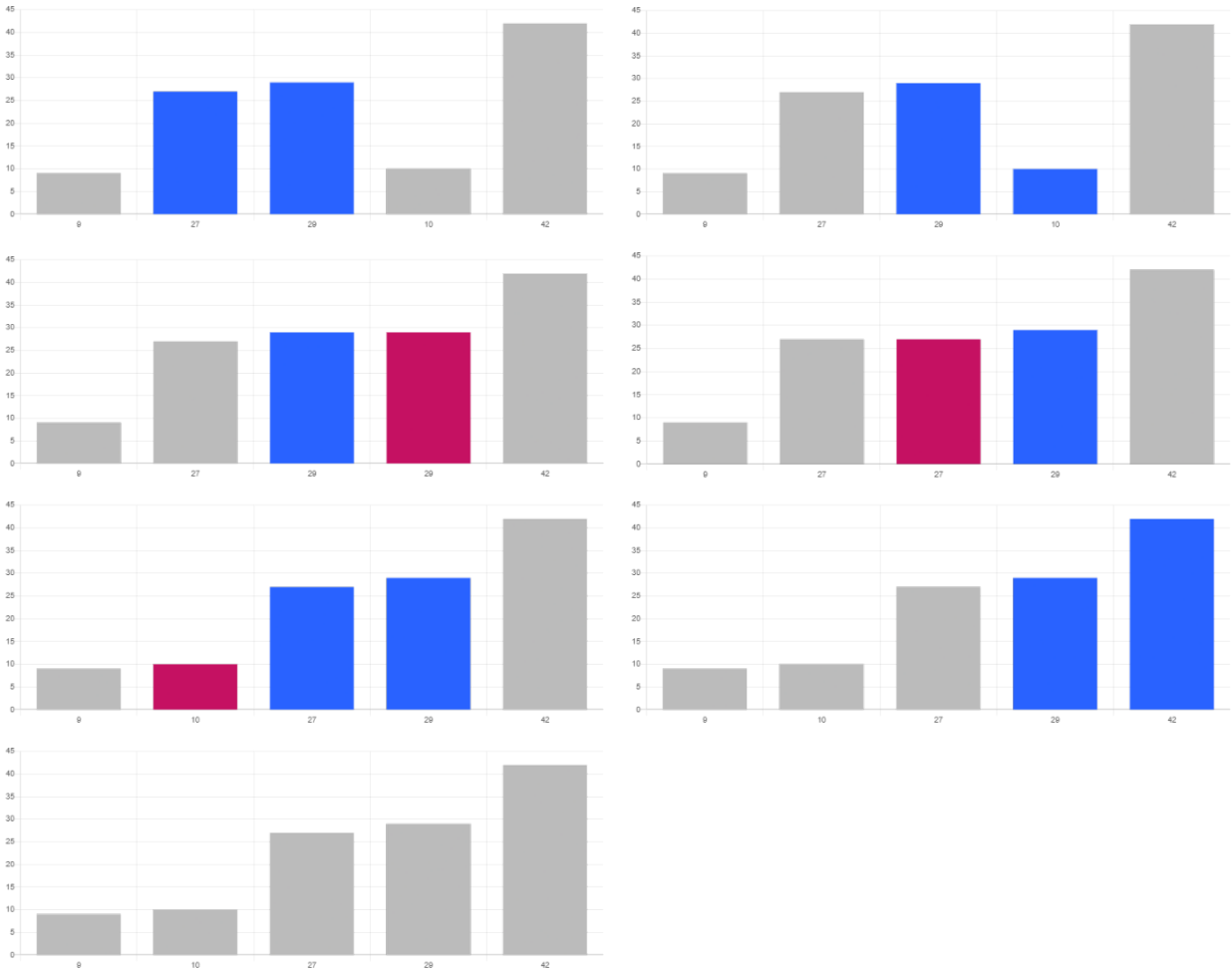
Link para revisar el Sort animado: https://algorithm-visualizer.org/divide-and-conquer/quicksort

Codigo Ejecutado:

```java
import org.algorithm_visualizer.*;

import java.util.Arrays;

class Main {

    private static ChartTracer chartTracer = new ChartTracer();
    private static LogTracer logTracer = new LogTracer("Console");
    private static Array1DTracer tracer = new Array1DTracer();

    // Arreglo fijo
    private static Integer[] array = {29, 10, 42, 9, 27};

    public static void main(String[] args) {
        tracer.set(array);
        tracer.chart(chartTracer);
        Layout.setRoot(new VerticalLayout(new Commander[]{chartTracer, tracer, logTracer}));
        logTracer.printf("Original array = %s\n", Arrays.toString(array));

        Tracer.delay();
        quickSort(array, 0, array.length - 1);
        logTracer.printf("Sorted array = %s\n", Arrays.toString(array));
    }

    public static void quickSort(Integer[] arr, int left, int right) {
        int l, r, s;
        while (right > left) {
            l = left;
            r = right;
            s = arr[left];
            while (l < r) {
                tracer.select(left);
                tracer.select(right);
                Tracer.delay();

                while (arr[r] > s) {
                    tracer.select(r);
                    Tracer.delay();
                    tracer.deselect(r);
                    r--;
                }
                arr[l] = arr[r];
                tracer.patch(l, arr[r]);
                Tracer.delay();
                tracer.depatch(l);

                while (s >= arr[l] && l < r) {
                    tracer.select(l);
                    Tracer.delay();
                    tracer.deselect(l);
                    l++;
                }
                arr[r] = arr[l];
                tracer.patch(r, arr[l]);
                Tracer.delay();
                tracer.depatch(r);
                tracer.deselect(left);
                tracer.deselect(right);
            }
            arr[l] = s;
            tracer.patch(l, s);
            Tracer.delay();
            tracer.depatch(l);
            quickSort(arr, left, l - 1);
            left = l + 1;
        }
    }
}
```

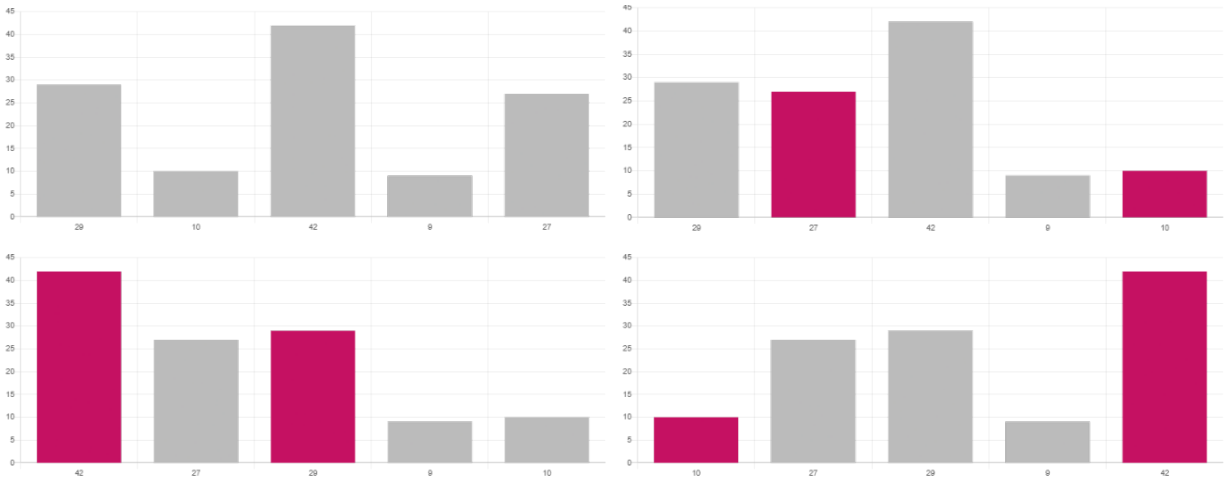
1.3.5. Shell Sort.

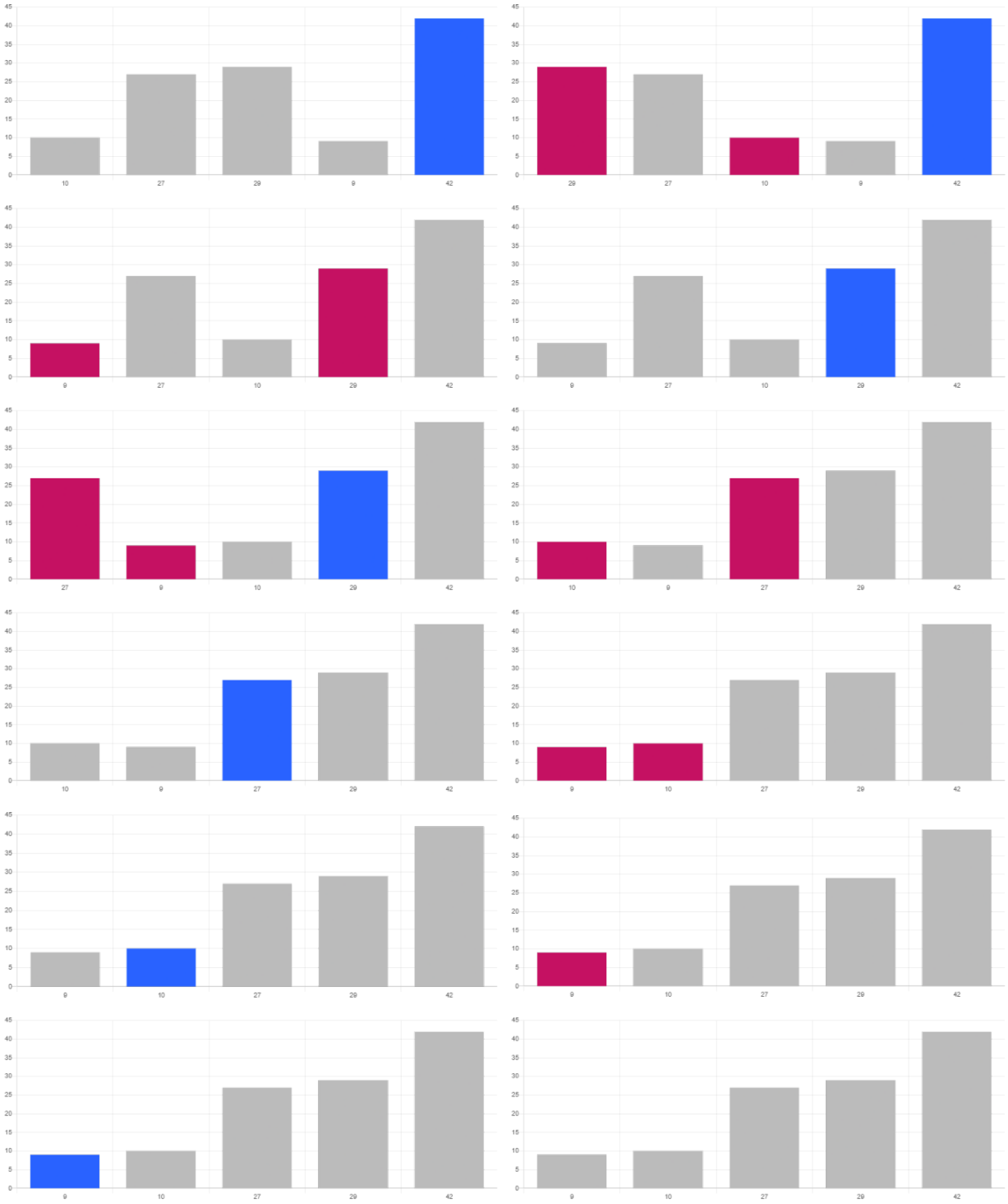




Link para revisar el Sort animado: https://algorithm-visualizer.org/brute-force/shellsort

1.3.6. Heap Sort.





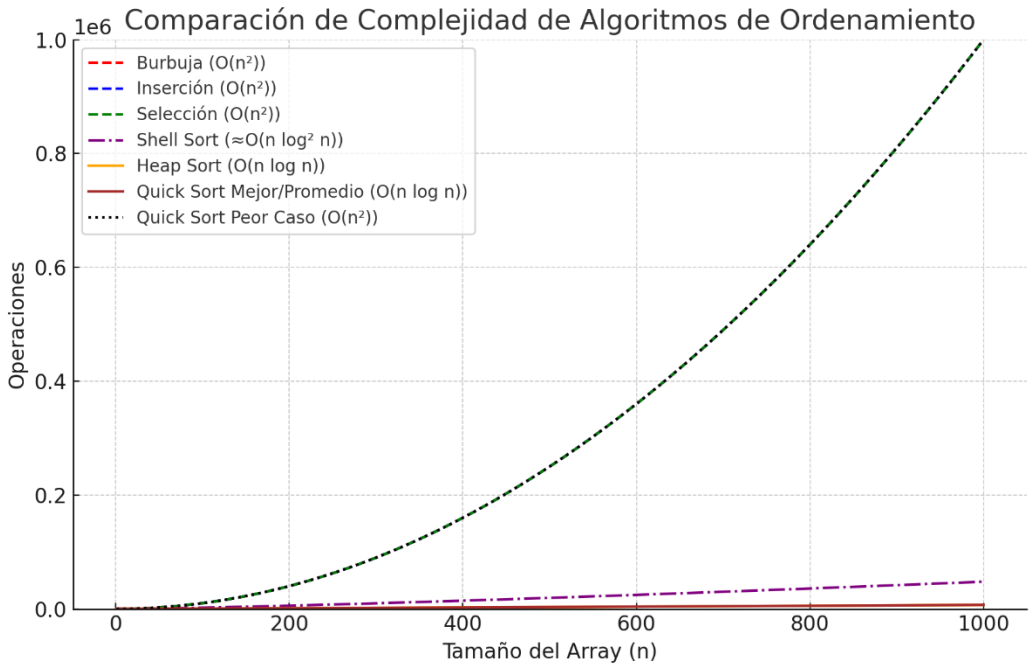
Link para revisar el Sort animado: https://algorithm-visualizer.org/brute-force/heapsort

1.4.Explicación grafica de la complejidad algorítmica.

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso
Burbuja	$O(n)$	$O(n^2)$	$O(n^2)$
Inserción	$O(n)$	$O(n^2)$	$O(n^2)$
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(n \log n)$	Depende del gap $O\left(\frac{n^3}{n^2}\right)$ o $O(n\,log^2\,n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Expresiones	Explicación
$\boldsymbol{O(1)}$	• No importa el tamaño de la entrada, siempre toma el mismo tiempo. • Ejemplo: Acceder a un elemento de un array por su índice → $array[0]$
$\boldsymbol{O(n)}$	• El tiempo de ejecución crece de manera proporcional a n • Ejemplo: Recorrer un array una vez.
$\boldsymbol{O(n^2)}$	• Si el tamaño de la entrada se duplica, el tiempo de ejecución se cuadruplica. • Ocurre cuando se usan dos ciclos anidados.
$\boldsymbol{O(n \log n)}$	• Ocurre en algoritmos de "divide y vencerás" como Merge Sort y Quick Sort (en el mejor y promedio caso). • Ejemplo: Quick Sort, Heap Sort.
$\boldsymbol{O(\log n)}$	• Se da cuando el tamaño del problema se reduce a la mitad en cada paso.

	• Ejemplo: Búsqueda binaria en un array ordenado.



1.4.1. Burbuja (Bubble Sort)

- Mejor Caso: $O(n)$→ Cuando el arreglo ya está ordenado, solo hace una pasada sin intercambios.
- Caso Promedio y Peor Caso: $O(n^2)$→ En el peor de los casos (arreglo en orden inverso), tiene que hacer muchas comparaciones e intercambios.

Ineficiente para grandes volúmenes de datos.

1.4.2. Inserción (Insertion Sort)

- Mejor Caso: $O(n)$→ Si el arreglo ya está ordenado, solo revisa cada elemento una vez.
- Caso Promedio y Peor Caso: $O(n^2)$→ Cuando el arreglo está desordenado, tiene que comparar e insertar cada elemento en su posición correcta.

Bueno para listas pequeñas o casi ordenadas.

1.4.3. Selección (Selection Sort)

- Siempre: $O(n^2)\rightarrow$ → No importa si el arreglo está ordenado o no, siempre busca el mínimo en cada iteración y lo intercambia.

Menos intercambios que Burbuja, pero sigue siendo ineficiente para grandes conjuntos de datos.

1.4.4. Shell Sort

- Mejor Caso: $O(n \log n)\rightarrow$ Depende del tamaño del "gap" usado.
- Caso Promedio: Depende del gap $O\left(\frac{n^3}{n^2}\right)$ o $O(n\,log^2\,n)\rightarrow$ No tiene una complejidad promedio bien definida.
- Peor Caso: $O(n^2)$ → Si el gap no es bien elegido, se reduce a un ordenamiento por inserción ineficiente.

Mejora el Insertion Sort al reducir la cantidad de desplazamientos.

1.4.5. Heap Sort

- Siempre: $O(n \log n)\rightarrow$ Utiliza un heap binario para ordenar los elementos de manera eficiente en cualquier caso.

Rendimiento estable, pero con más operaciones en memoria por la reestructuración del heap.

1.4.6. Quick Sort

- Mejor Caso y Caso Promedio: $O(n \log n)\rightarrow$ → Si se elige bien el pivote, el problema se divide en dos partes equilibradas.
- Peor Caso: $O(n^2)$ → Si el pivote siempre es el menor o el mayor, se generan particiones desbalanceadas.

Es uno de los más rápidos en la práctica, pero su rendimiento depende de la elección del pivote.

2. Presaberes requeridos

2.1.¿Cómo adiciona datos en un arreglo de nombres, de tamaño 10 desde su creación? Escriba el algoritmo en Java

```java
import java.util.Scanner;

public class NombresArray {
    public static void main(String[] args) {
        String[] nombres = new String[10]; // Crear un arreglo de tamaño 10
        Scanner scanner = new Scanner(System.in);

        // Llenar el arreglo con nombres
        for (int i = 0; i < nombres.length; i++) {
            System.out.print("Ingrese el nombre " + (i + 1) + ": ");
            nombres[i] = scanner.nextLine(); // Leer el nombre desde la entra
        }

        // Mostrar los nombres ingresados
        System.out.println("Nombres ingresados:");
        for (String nombre : nombres) {
            System.out.println(nombre);
        }

        scanner.close();
    }
}
```

2.2.¿Cómo compararía dos datos tipo String en Java? Describa el procedimiento.

```java
String str1 = "Hola";
String str2 = "Hola";

if (str1.equals(str2)) {
    System.out.println("Las cadenas son iguales.");
} else {
    System.out.println("Las cadenas son diferentes.");
}
```

2.3.Se tiene dos variables que almacenan valores diferentes. Diseñe un algoritmo que le permita intercambiar los valores entre ellas.

Ejemplo:

var1 = 2;

var2 = 600;

Algoritmo para intercambiar valores entre dos variables

```plaintext
1. Iniciar
2. Definir var1 y var2 con valores iniciales
3. Crear una variable temporal (temp)
4. Asignar el valor de var1 a temp
5. Asignar el valor de var2 a var1
6. Asignar el valor de temp a var2
7. Fin
```

Ejemplo en pseudocódigo

```
var1 = 2
var2 = 600

// Intercambio
temp = var1
var1 = var2
var2 = temp

// Después del intercambio
Imprimir var1 // 600
Imprimir var2 // 2
```

3. **Preguntas orientadoras**

3.1. Suponga que usted tiene que ordenar la siguiente lista de números: [15, 5, 4, 18, 12, 19, 14, 10, 8, 20] ¿Cuál de las siguientes listas representa la lista parcialmente ordenada después de tres pasadas completas del ordenamiento por inserción?:

a. [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]

b. [15, 5, 4, 10, 12, 8, 14, 18, 19, 20]

c. [4, 5, 15, 18, 12, 19, 14, 10, 8, 20]

d. [15, 5, 4, 18, 12, 19, 14, 8, 10, 20]

Respuesta

Ordenamiento por inserción: La lista parcialmente ordenada después de tres pasadas completas es:

- a. [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]

Explicación: En el ordenamiento por inserción, se toma un elemento y se inserta en su posición correcta en la parte ya ordenada de la lista. Después de tres pasadas, los primeros tres elementos (4, 5, 15) están en su lugar correcto.

3.2. Suponga que usted tiene que ordenar la siguiente lista de números: [11, 7, 12, 14, 19, 1, 6, 18, 8, 20] ¿Cuál de las siguientes listas representa la lista parcialmente ordenada después de tres pasadas completas del ordenamiento por selección?

a. [7, 11, 12, 1, 6, 14, 8, 18, 19, 20]

b. [7, 11, 12, 14, 19, 1, 6, 18, 8, 20]

c. [11, 7, 12, 14, 1, 6, 8, 18, 19, 20]

d. [11, 7, 12, 14, 8, 1, 6, 18, 19, 20]

Respuesta

Ordenamiento por selección: La lista parcialmente ordenada después de tres pasadas completas es:

- a. [7, 11, 12, 1, 6, 14, 8, 18, 19, 20]

Explicación: En el ordenamiento por selección, se selecciona el menor elemento de la lista y se coloca al principio. Después de tres pasadas, los tres primeros elementos (7, 11, 12) están en su lugar correcto.

3.3. ¿Cuál es la búsqueda más eficiente en grandes cantidades de datos ya ordenados?

a. Binaria

b. Burbuja

c. Lineal

d. Quicksort

Respuesta.

- a. Binaria

Explicación: La búsqueda binaria es más eficiente en listas ordenadas, ya que reduce el espacio de búsqueda a la mitad en cada paso, lo que la hace mucho más rápida que la búsqueda lineal

3.4. Aprendizajes obtenidos y su relación con el futuro profesional:

- Comprensión de algoritmos de ordenamiento: Aprendimos cómo funcionan y se implementan distintos algoritmos de ordenamiento como Bubble Sort, Quick Sort y Merge Sort. Esto nos ayudará en nuestro futuro profesional al optimizar el manejo de datos en estructuras y bases de datos, mejorando la eficiencia de nuestras aplicaciones.
- Análisis de eficiencia y complejidad: Identificamos la importancia del análisis de la complejidad temporal y espacial de cada algoritmo. Esto es crucial en el desarrollo de software y optimización de sistemas, especialmente en entornos donde la velocidad y el uso de memoria son factores clave.
- Trabajo en equipo y resolución de problemas: Al enfrentar dificultades en la implementación, desarrollamos habilidades de comunicación y colaboración. En nuestra

carrera como ingenieros de software, será fundamental trabajar en equipo para resolver problemas de manera eficiente y estructurada.

3.5. Dificultades y estrategias de solución:

- Dificultad: La mayor dificultad fue entender cómo funcionan algunos algoritmos recursivos, como Merge Sort y Quick Sort, y cómo aplicar correctamente la recursión sin errores de pila o bucles infinitos.
- Estrategias de solución: Primero, analizamos cada algoritmo en papel, desglosando su flujo de ejecución paso a paso. Luego, utilizamos herramientas de depuración para identificar errores y mejorar nuestra comprensión. También buscamos documentación y ejemplos en línea para comparar implementaciones y optimizar nuestro código.