**GUÍA DE LABORATORIO NO. 3**

**HANNA KATHERINE ABRIL GÓNGORA**

**JENNIFER NATALIA BELTRAN**

**FELIPE ARAUJO**

**UNIVERSIDAD MANUELA BELTRÁN**

**ESTRUCTURA DE DATOS**

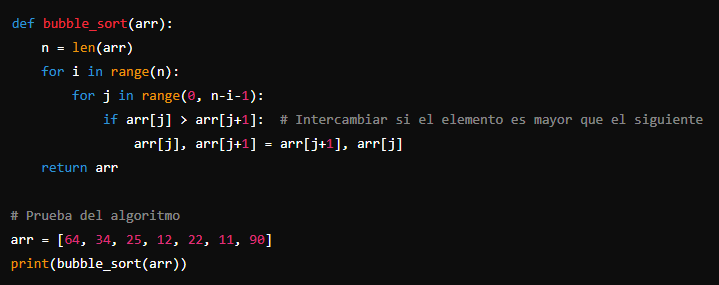
**DOCENTE**

**HUGO ALFONSO ORTIZ BARRERO**

**BOGOTA DC 24 FEBRERO 2025**

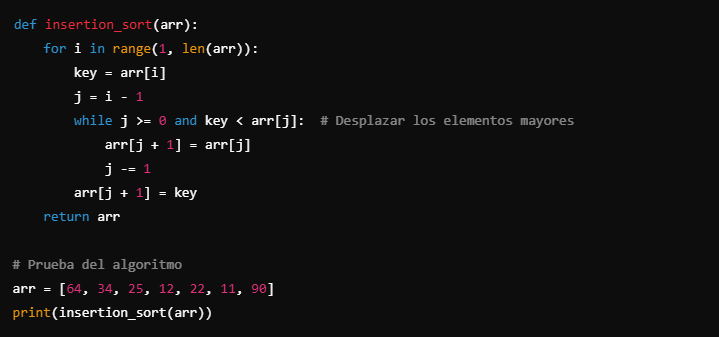
1. **Trabajo a Realizar.**
   1. Algoritmos de ordenamiento.
      1. Ordenamiento Burbuja (Bubble Sort)

El Bubble Sort es un algoritmo sencillo que compara elementos adyacentes y los intercambia si están en el orden incorrecto. Su complejidad es O(n²) en el peor caso.



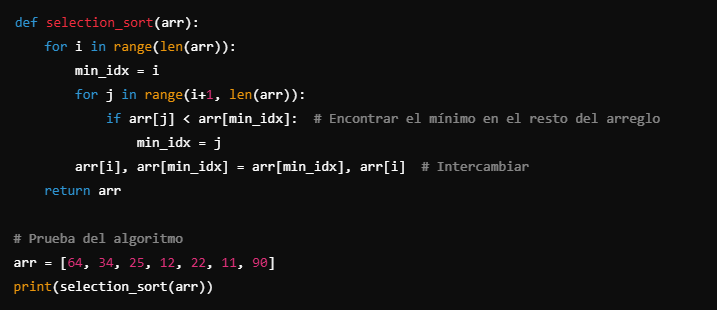
* + 1. Ordenamiento por Inserción (Insertion Sort)

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(n²).



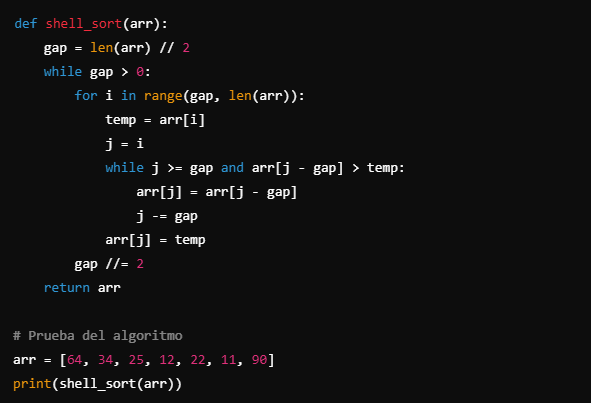
* + 1. Ordenamiento por Selección (Selection Sort)

El Selection Sort selecciona el elemento más pequeño y lo intercambia con el primer elemento no ordenado. Tiene una complejidad de O(n²).



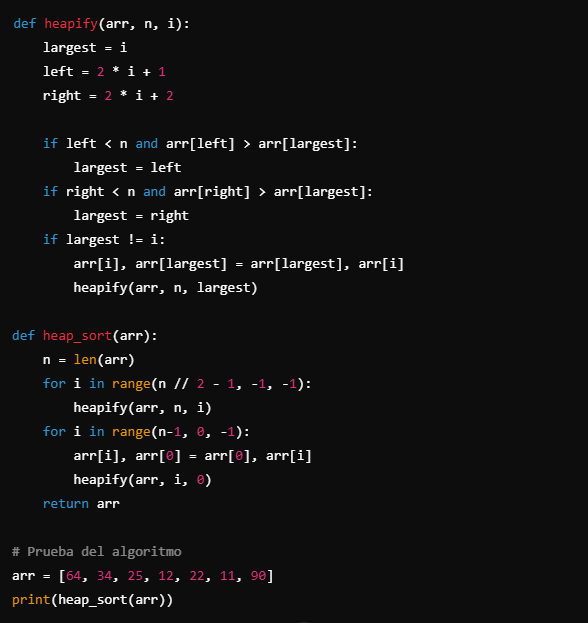
* + 1. Ordenamiento Shell (Shell Sort)

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).



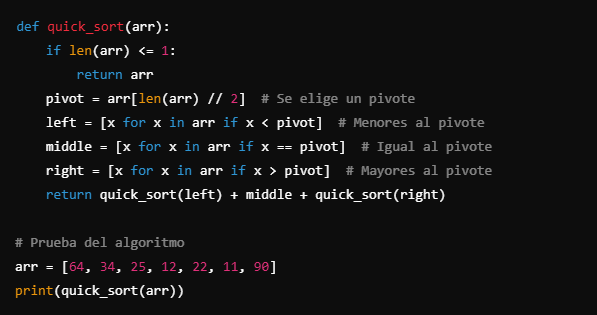
* + 1. Ordenamiento Heap (Heap Sort

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).



* + 1. 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(n²).



* 1. Pruebas de ordenado.
     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. 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.**

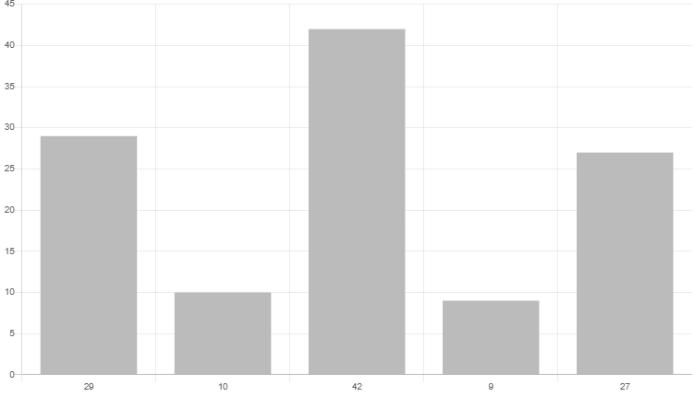
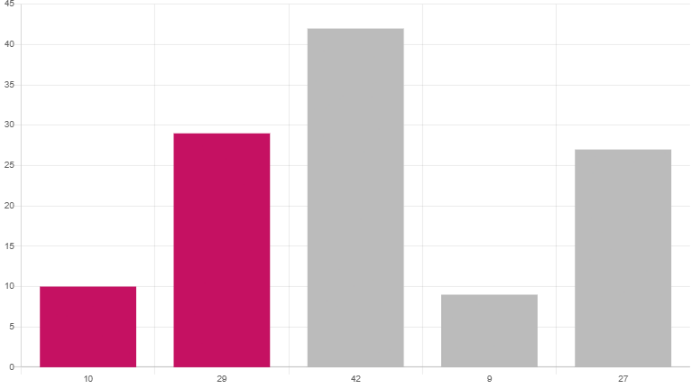
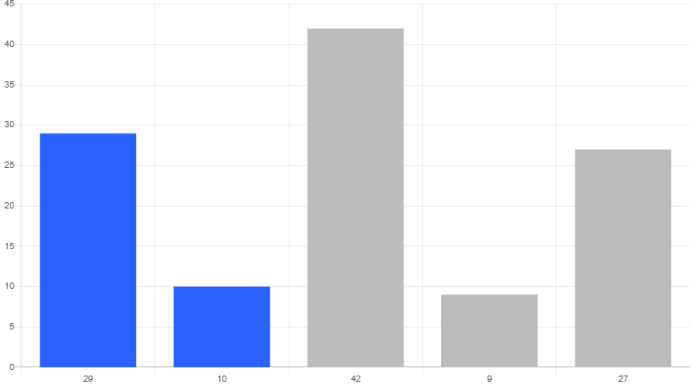
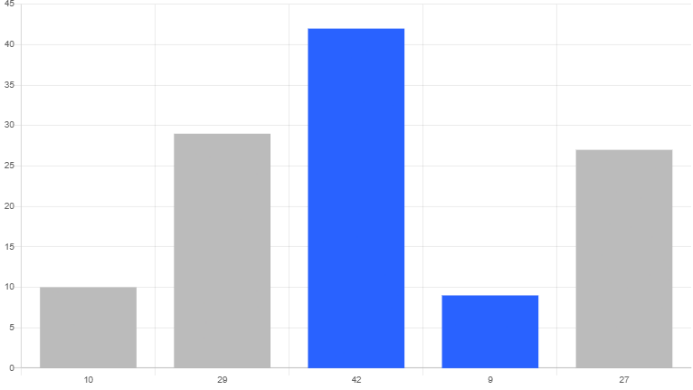
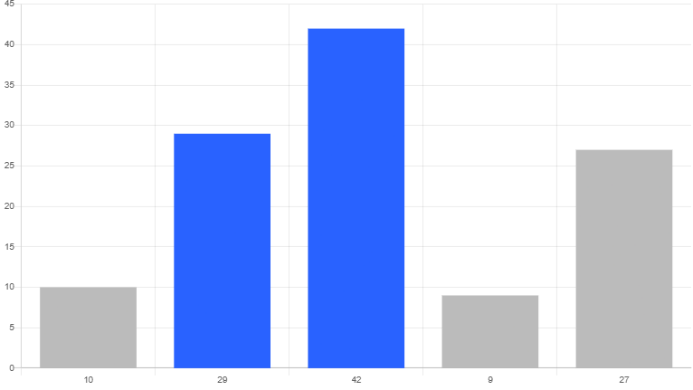
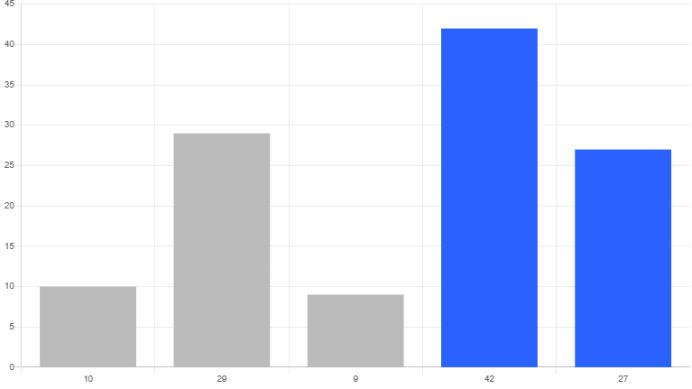
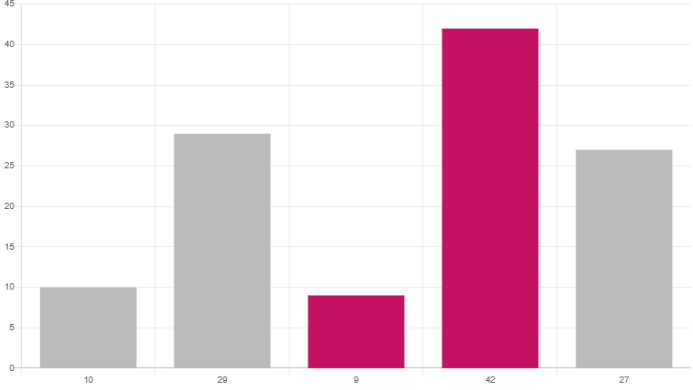
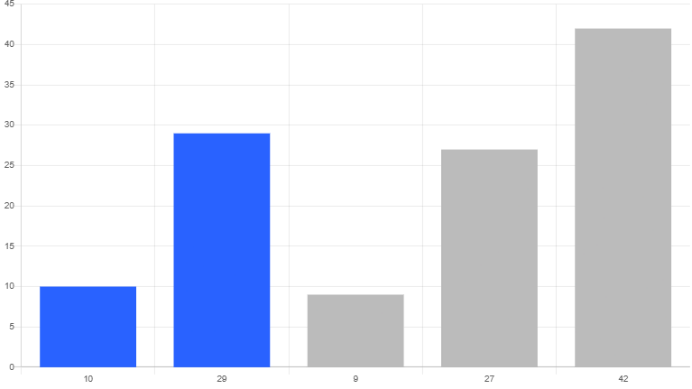
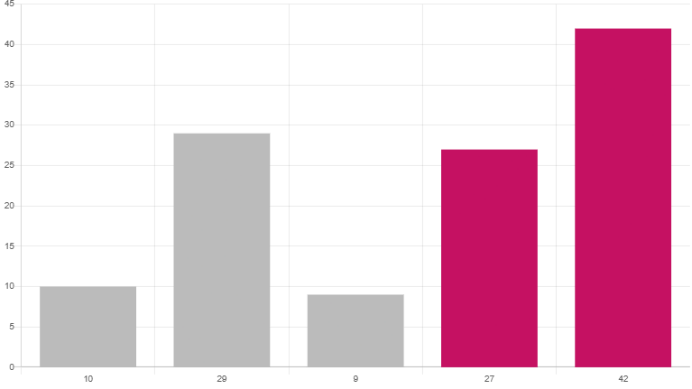
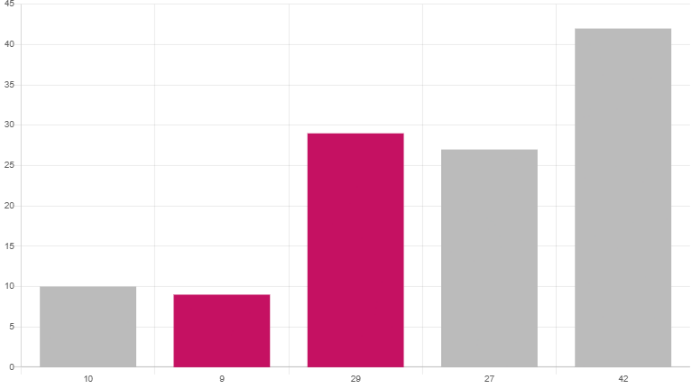
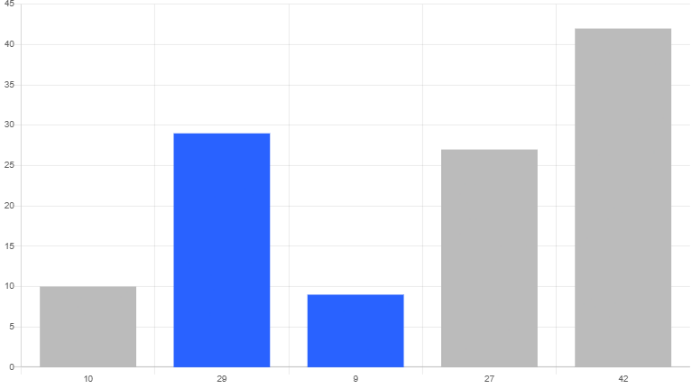
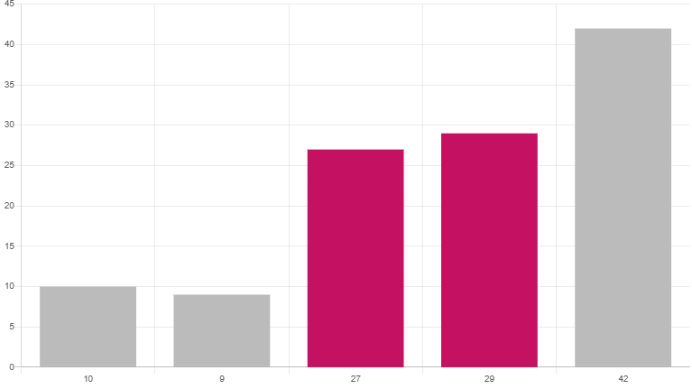
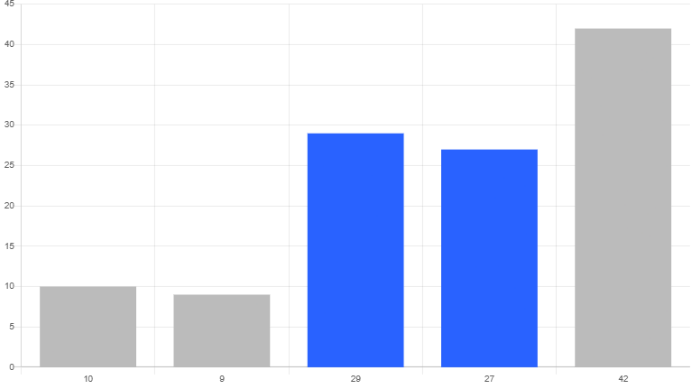
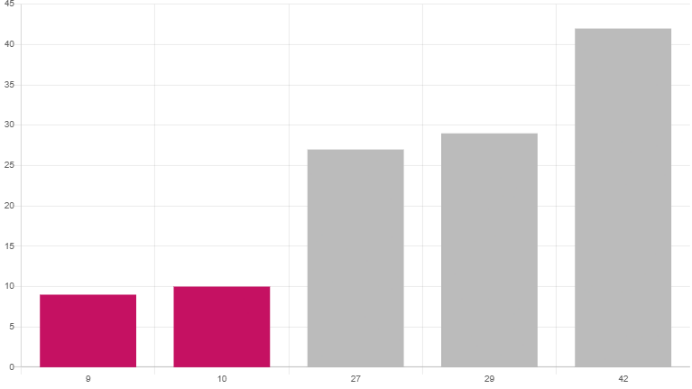
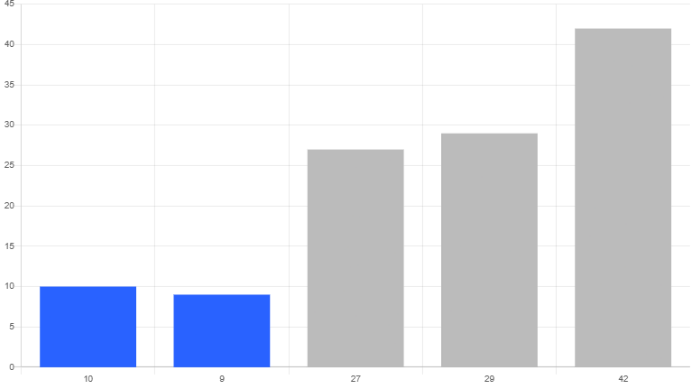
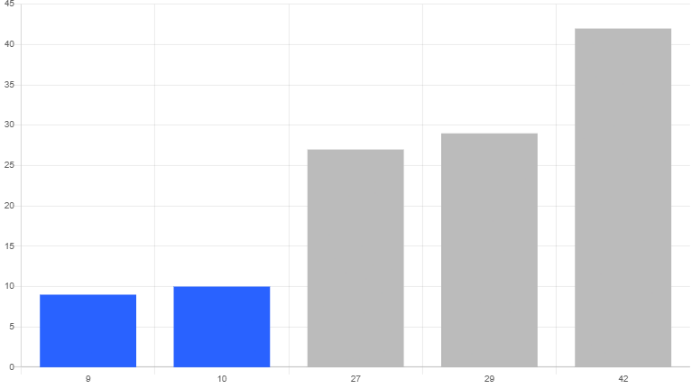
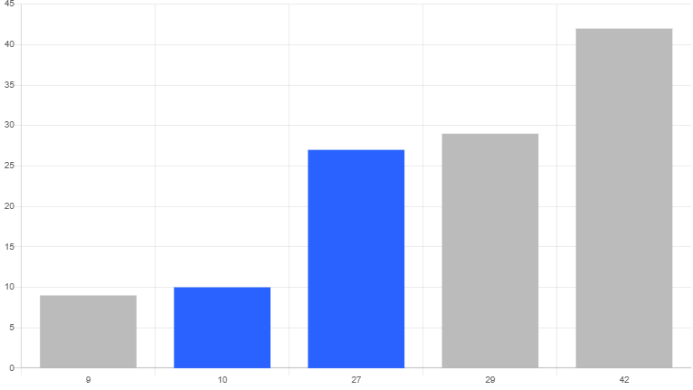
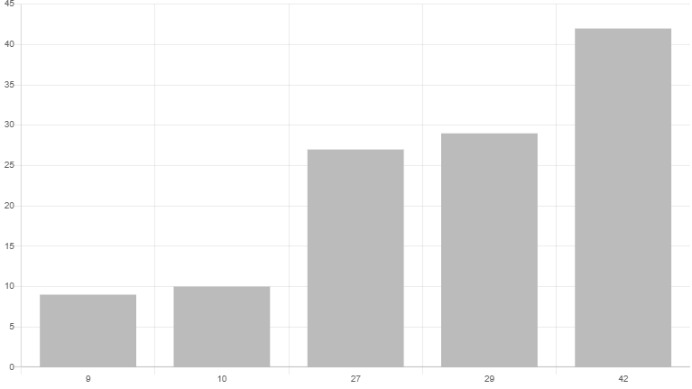
Insertion Sort toma cada elemento y lo coloca en la posición correcta dentro de la parte ordenada del arreglo. Se comparan los valores de derecha a izquierda hasta encontrar la posición adecuada.

|  |  |  |
| --- | --- | --- |
| 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. Explicación grafica de los algoritmos.

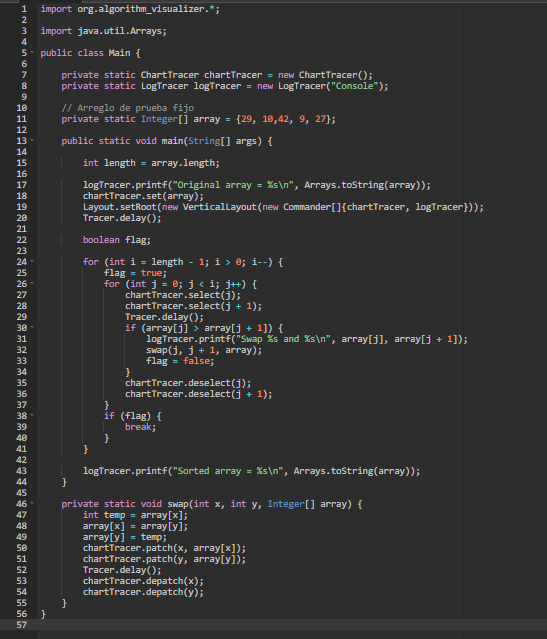
Arreglo de entrada: [29, 10,42, 9, 27]

* + 1. **Bubble Sort.**

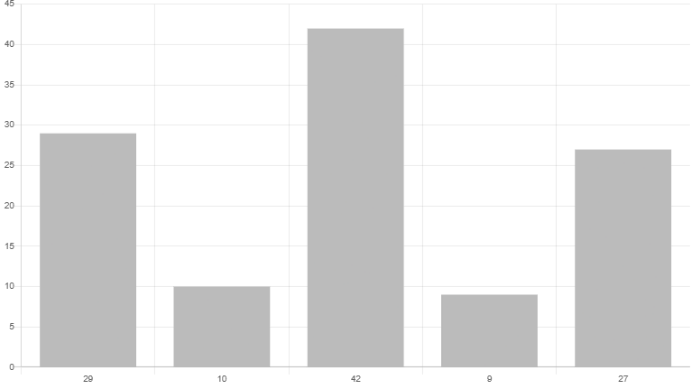
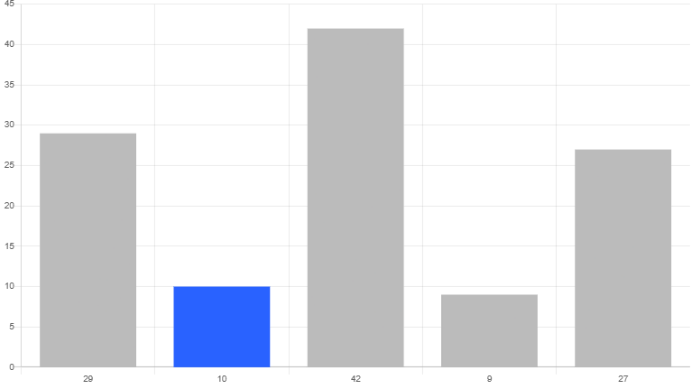
         

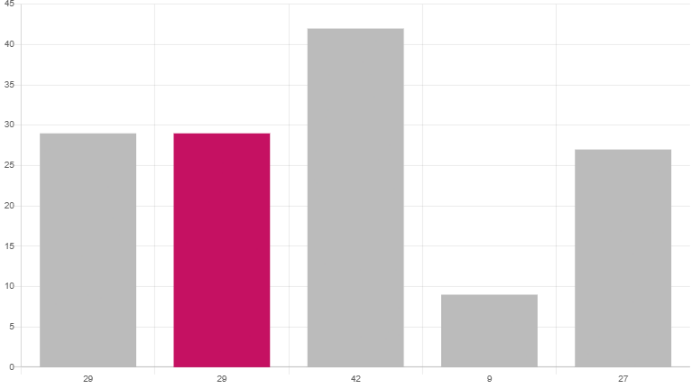
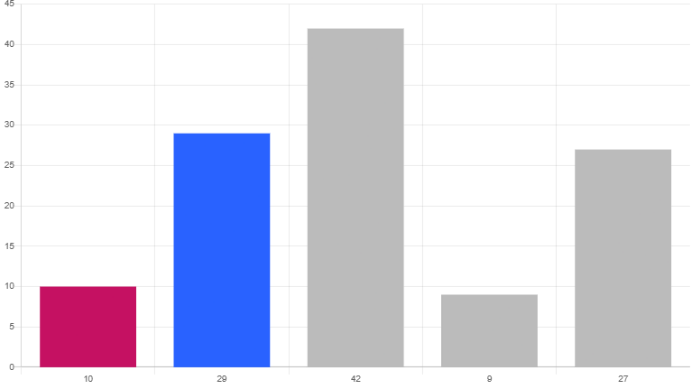
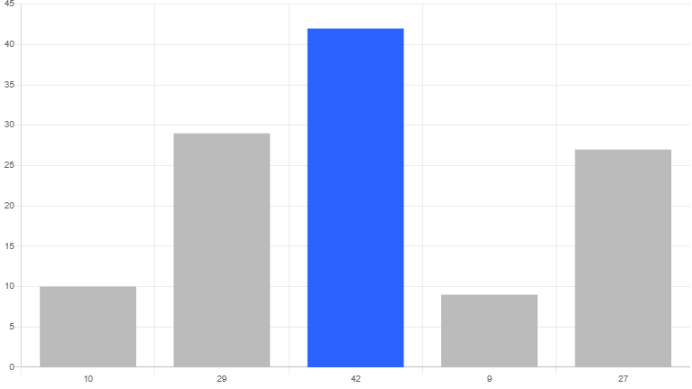
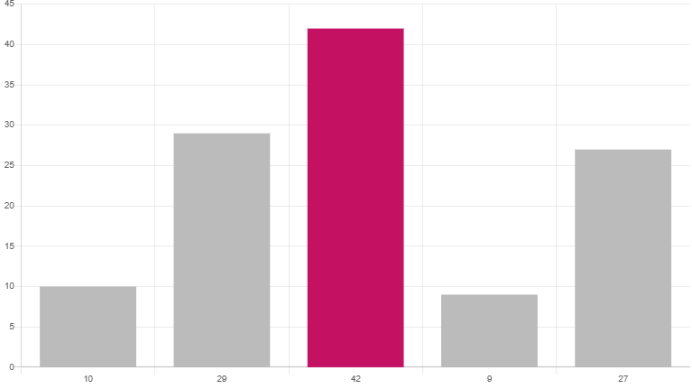
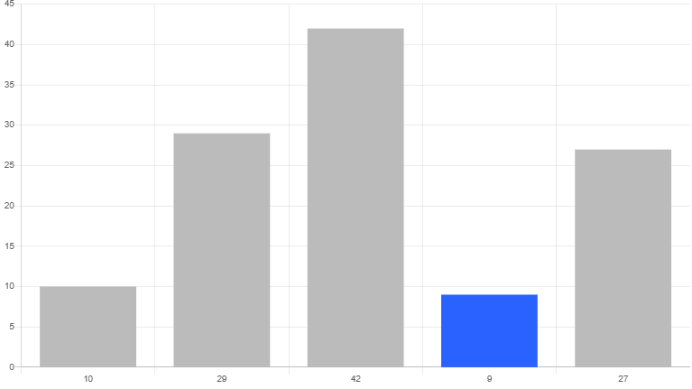
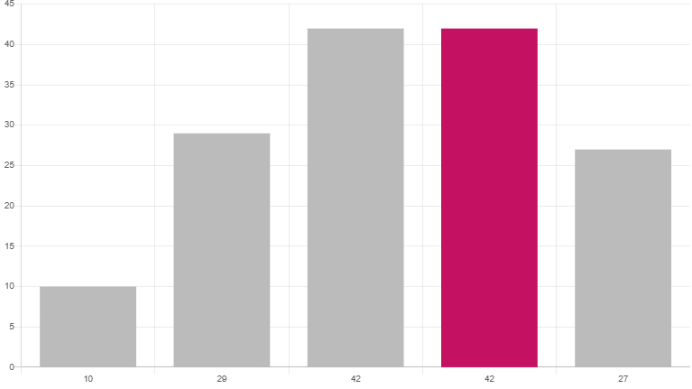
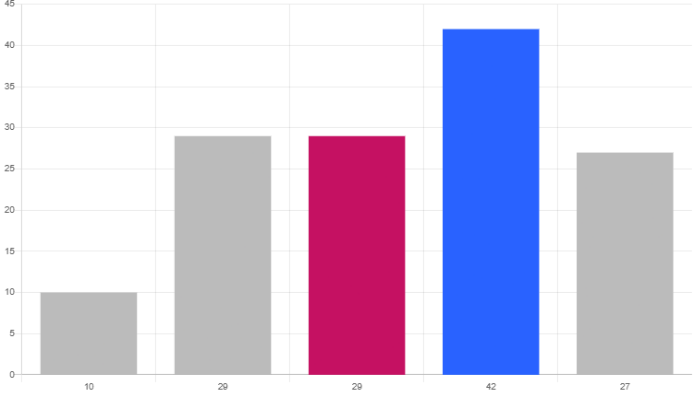
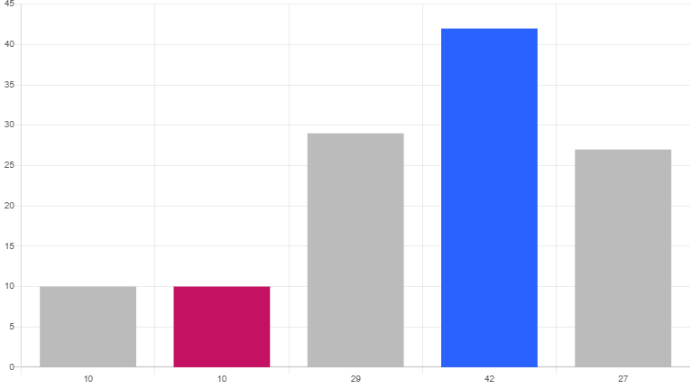
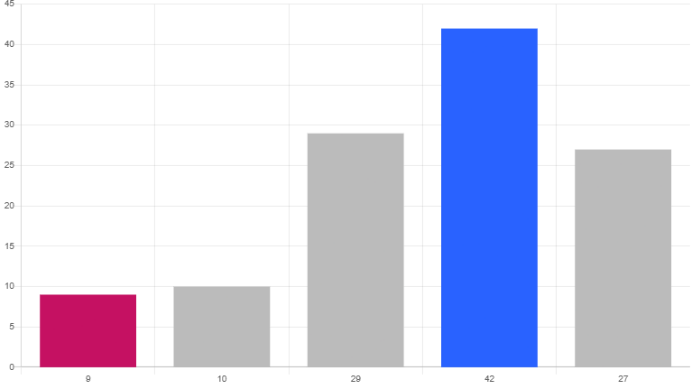
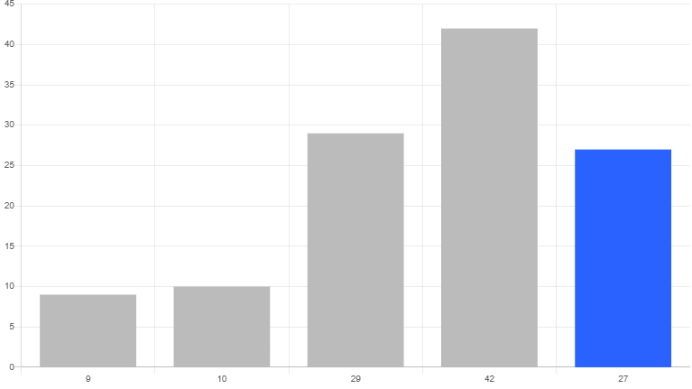
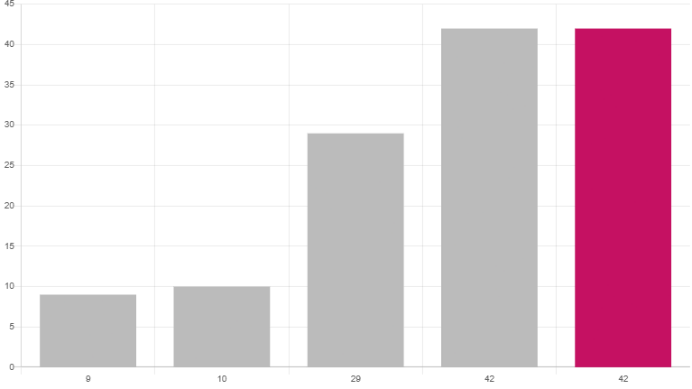
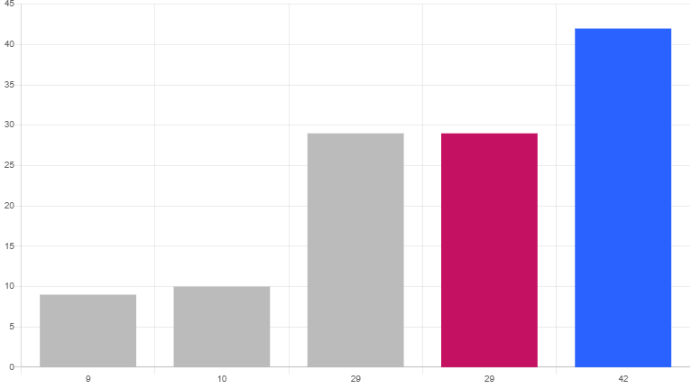
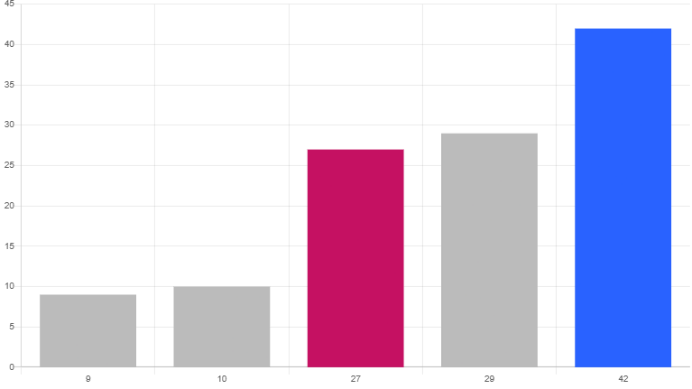
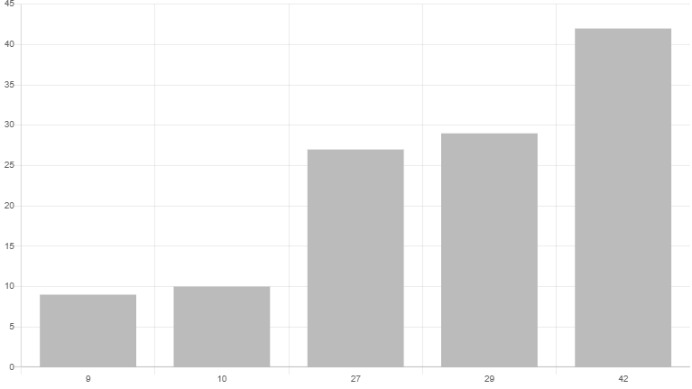
Link para revisar el Sort animado: <https://algorithm-visualizer.org/brute-force/bubble-sort>

Codigo Ejecutado:



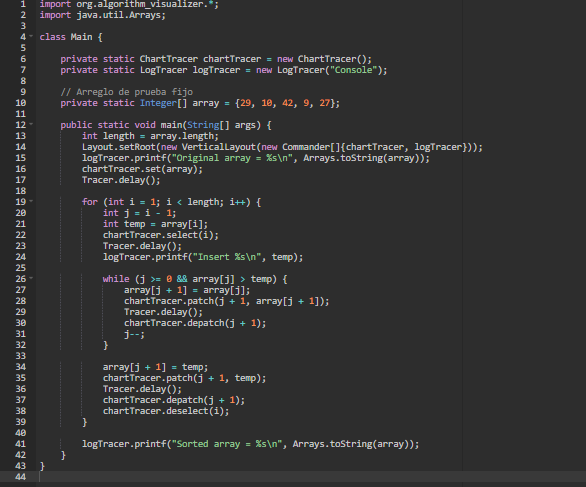
* + 1. **Insertion Sort.**

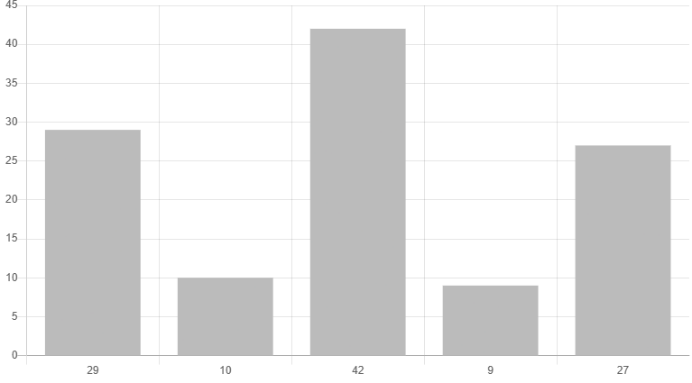
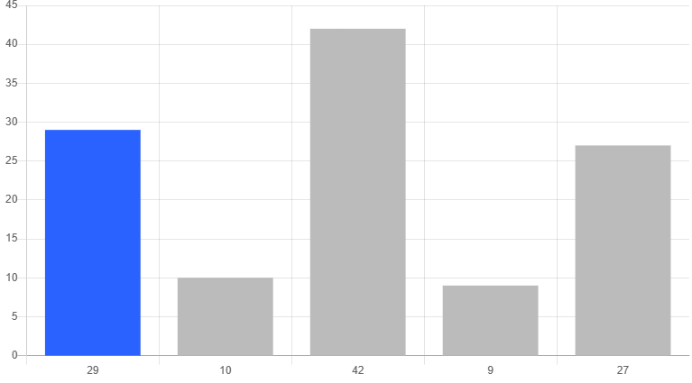
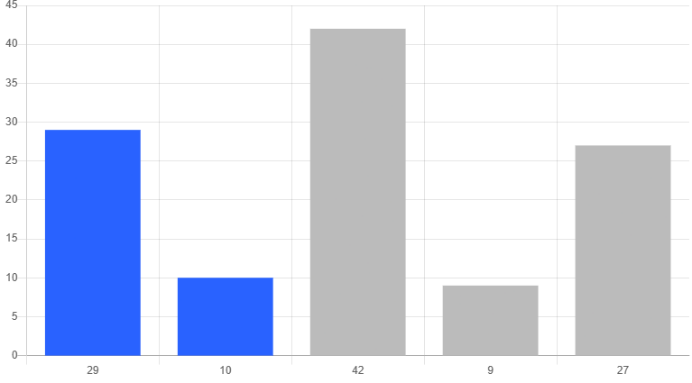
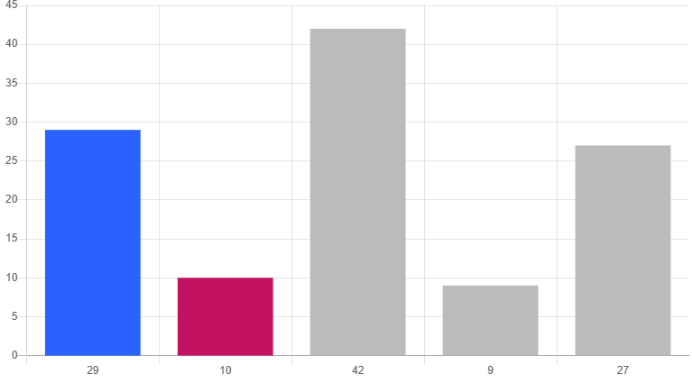
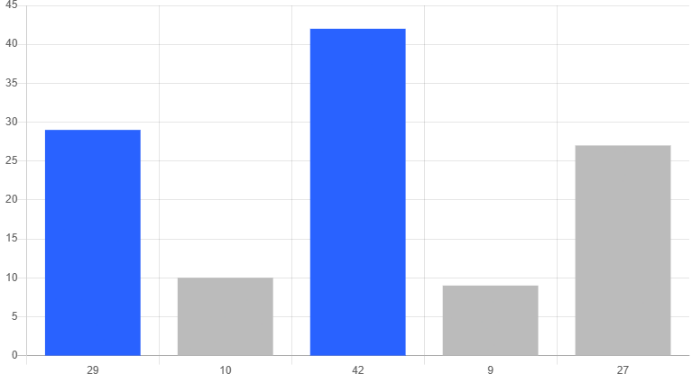
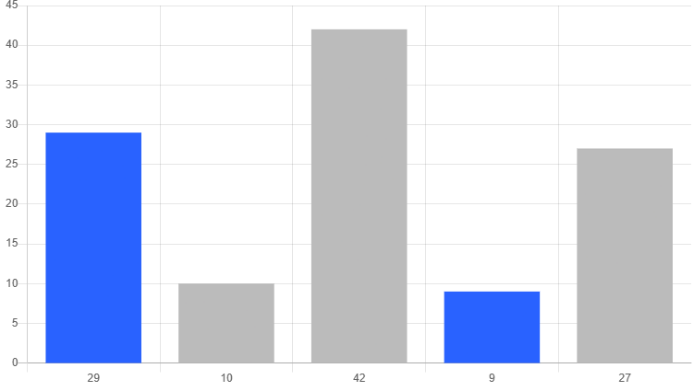
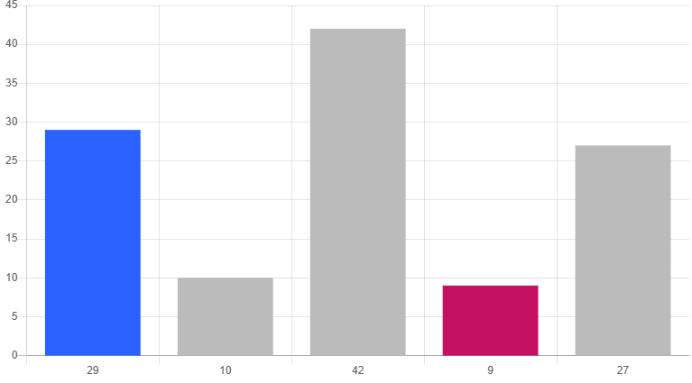
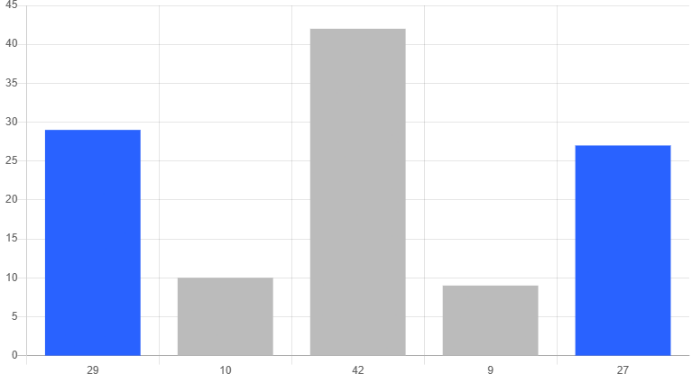
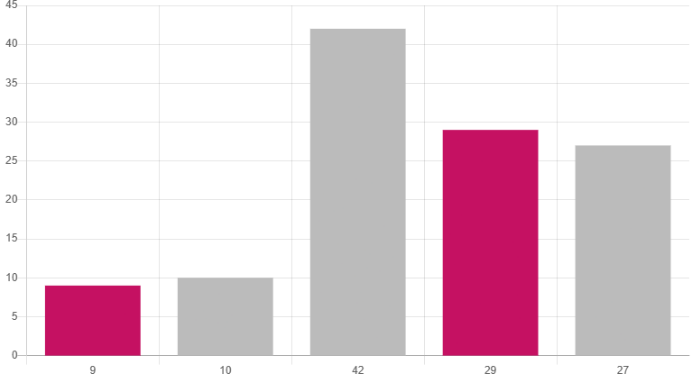
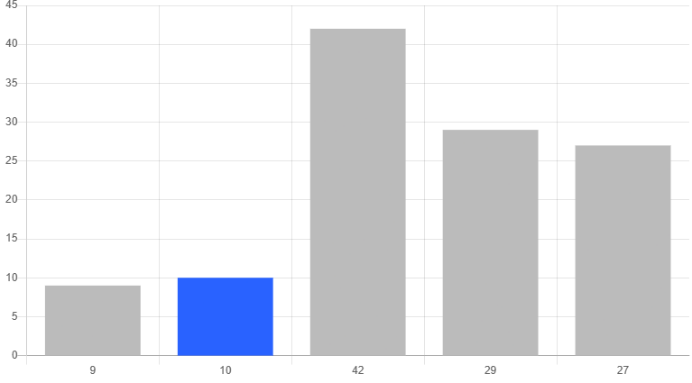
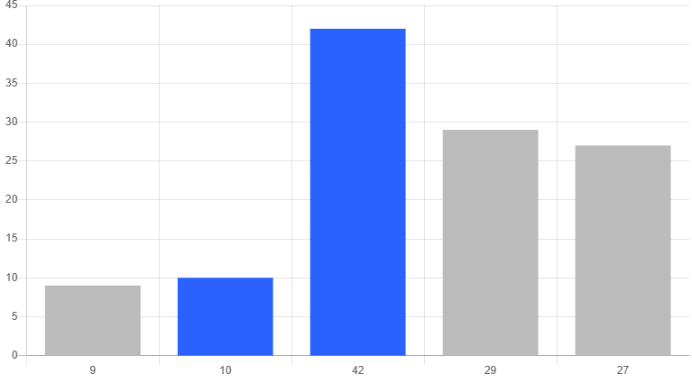
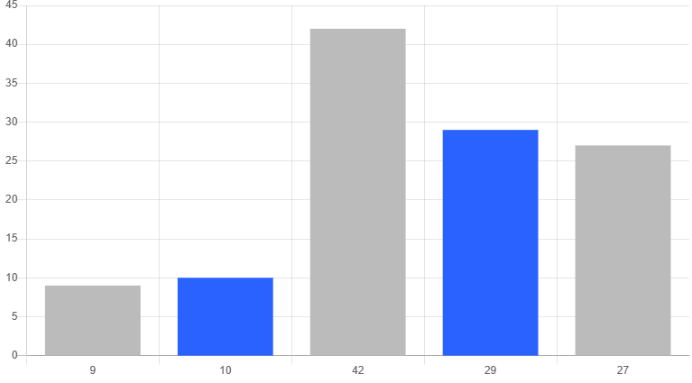
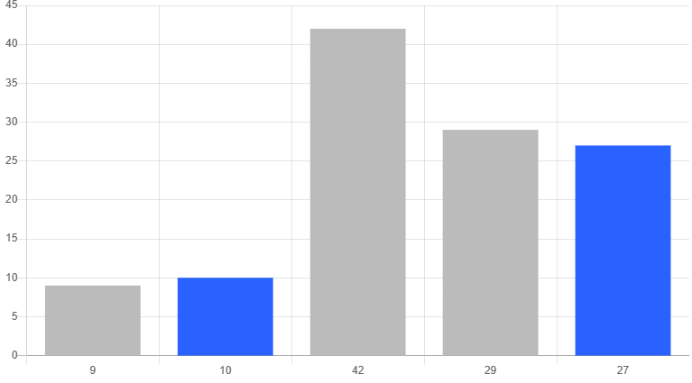
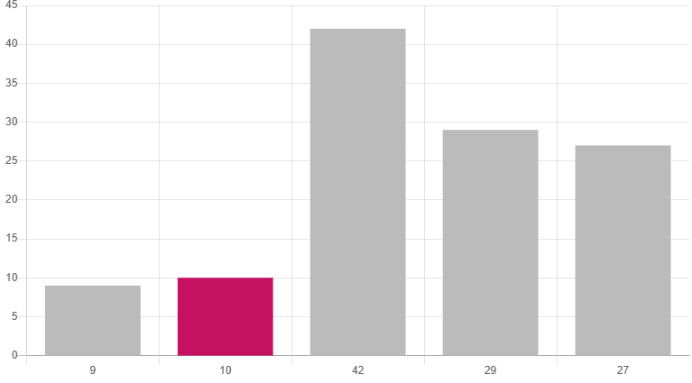
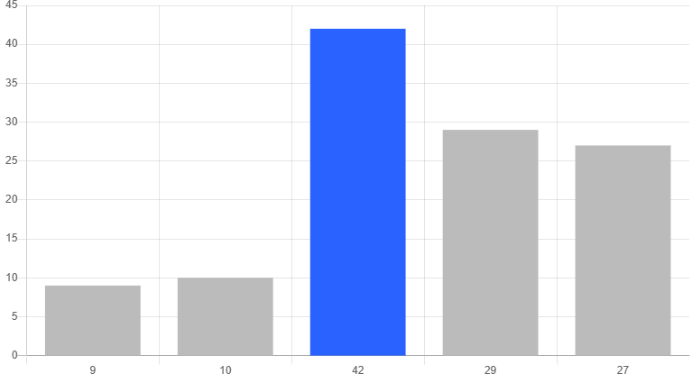
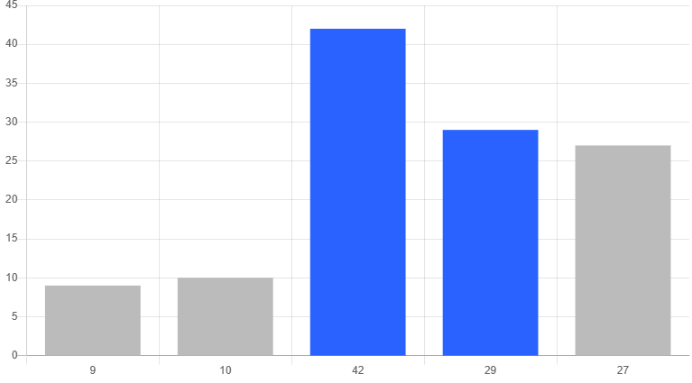
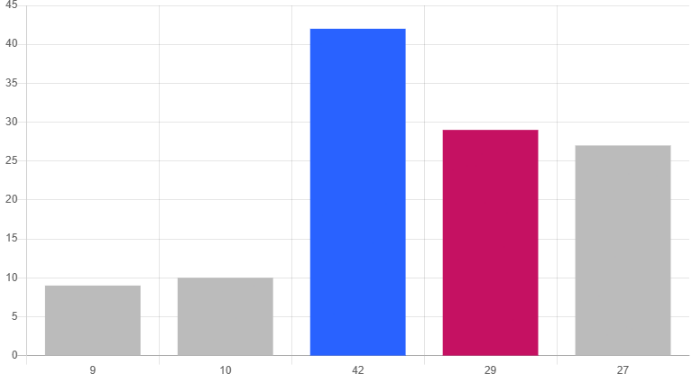
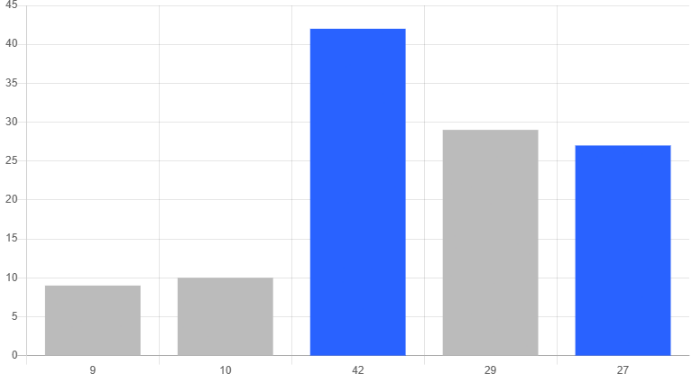
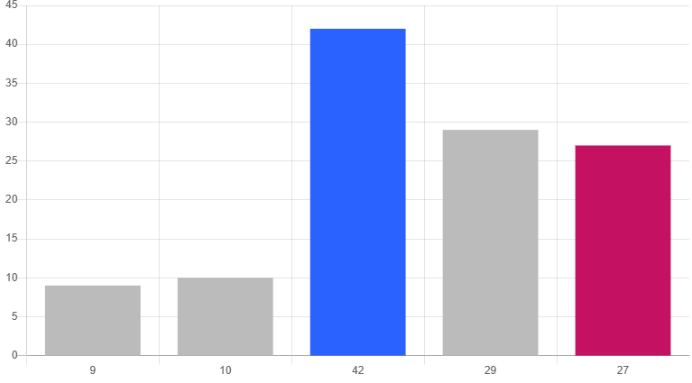
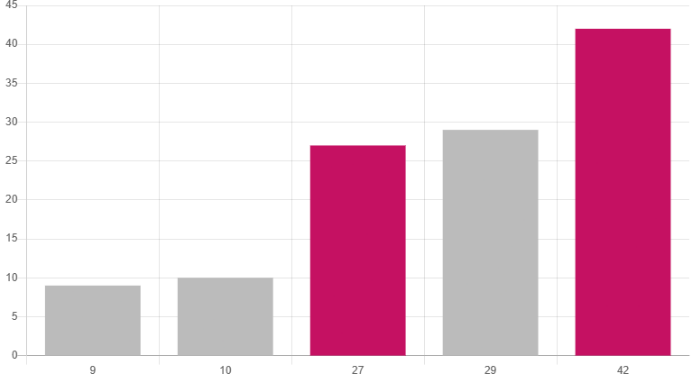
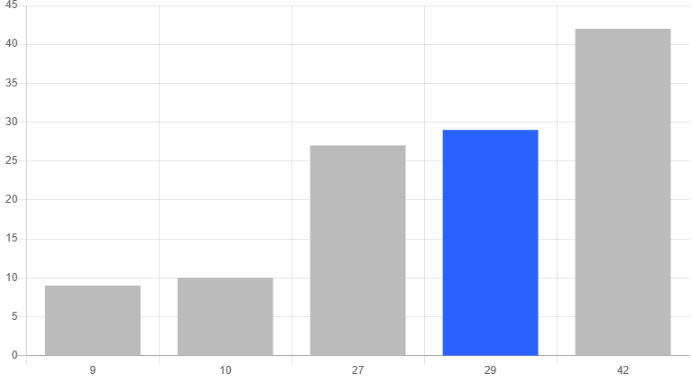
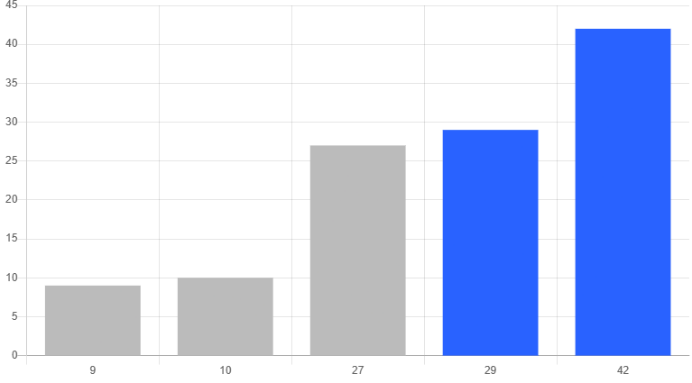
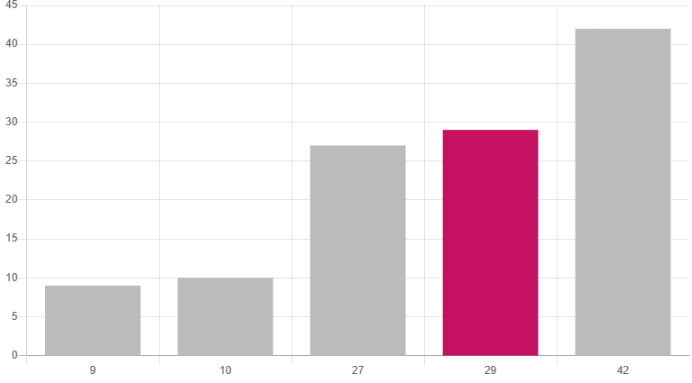
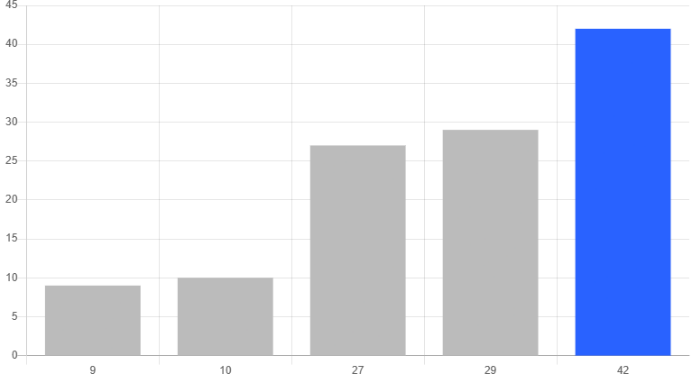
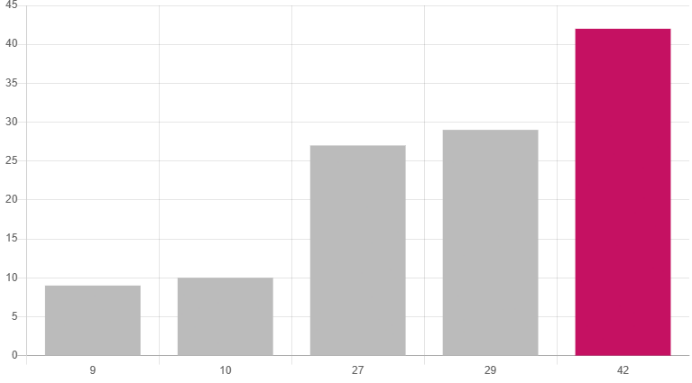
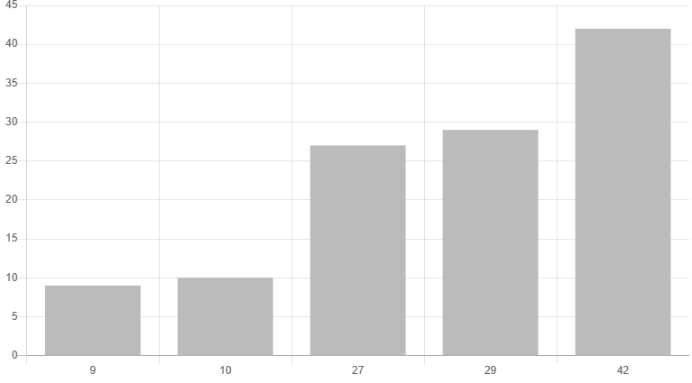
             

Link para revisar el Sort animado: <https://algorithm-visualizer.org/brute-force/insertion-sort>

Codigo Ejecutado:

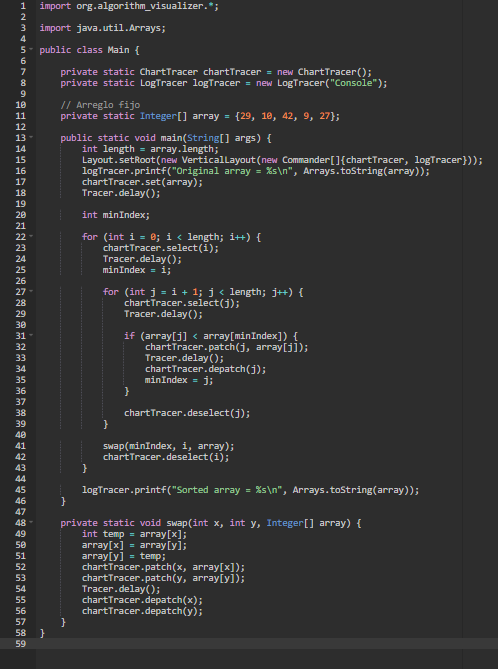


* + 1. **Selection Sort.**

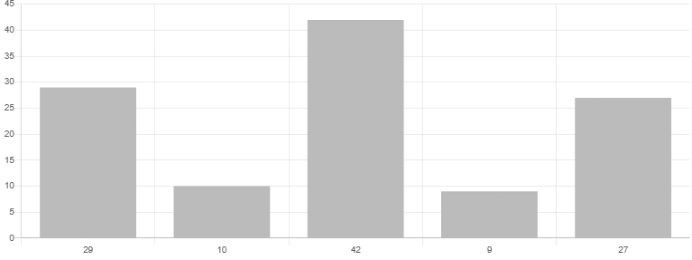
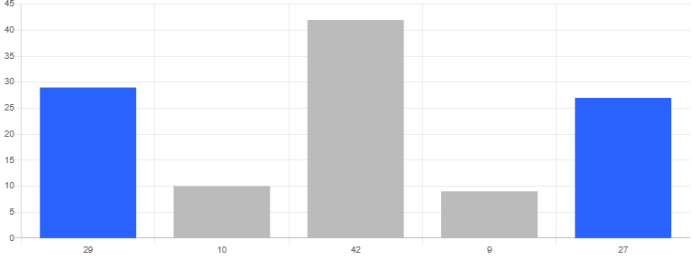
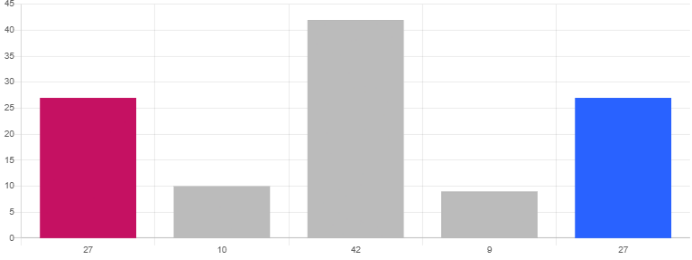
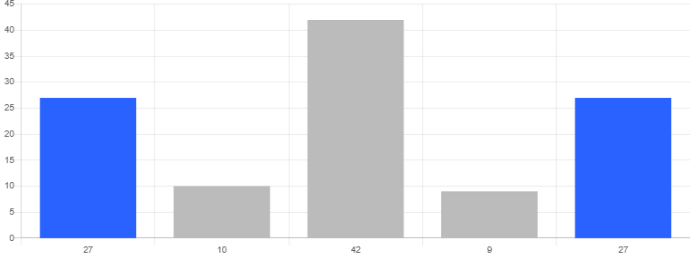
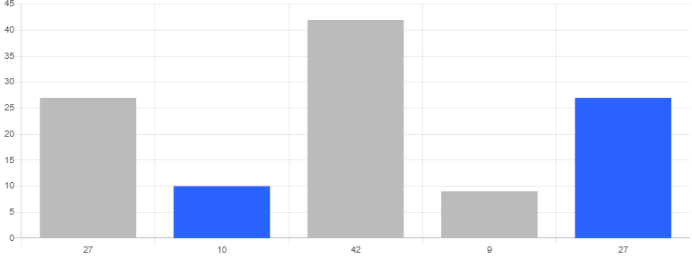
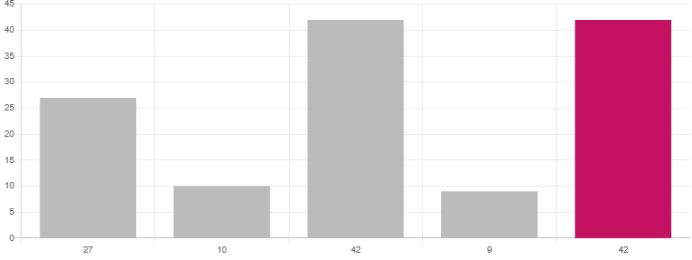
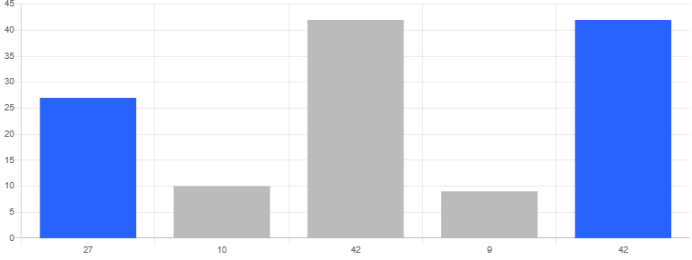
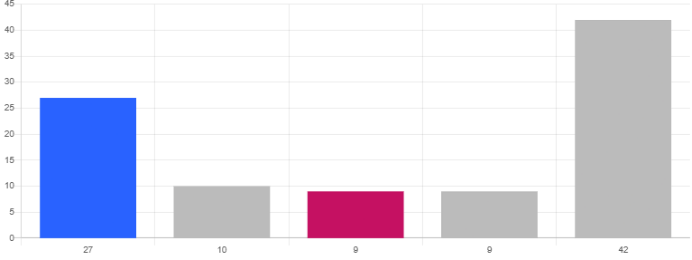
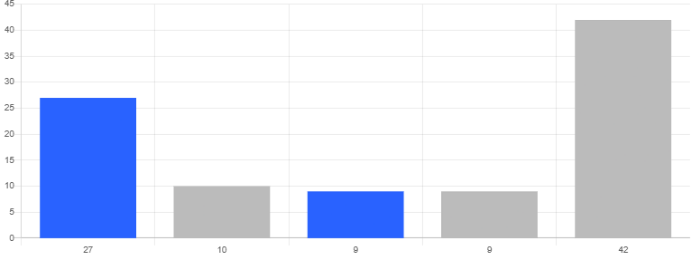
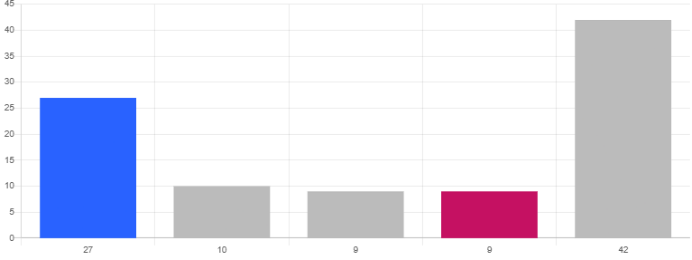
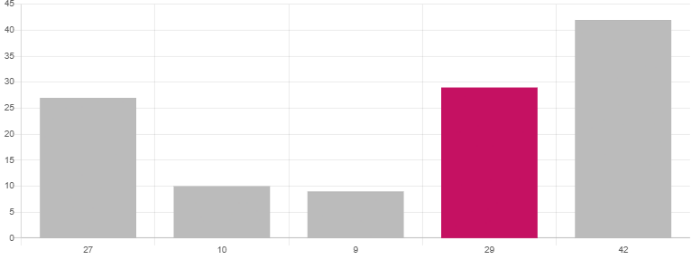
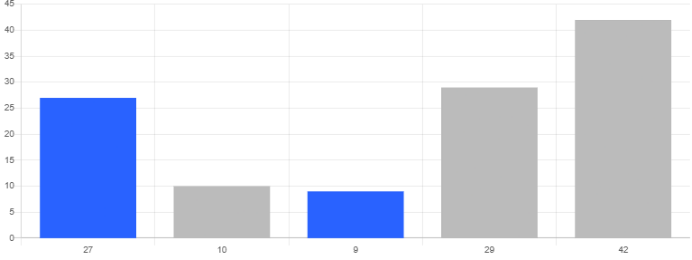
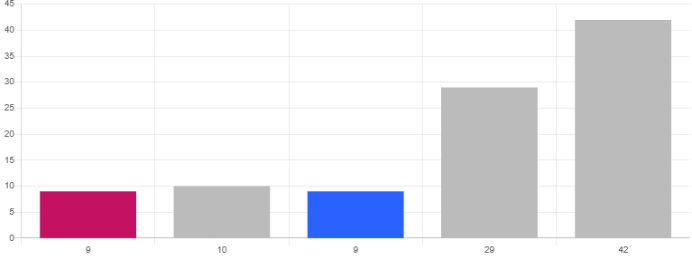
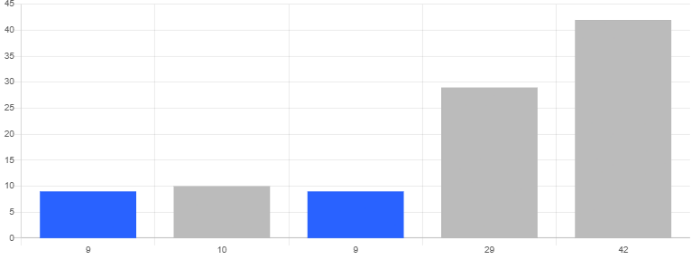
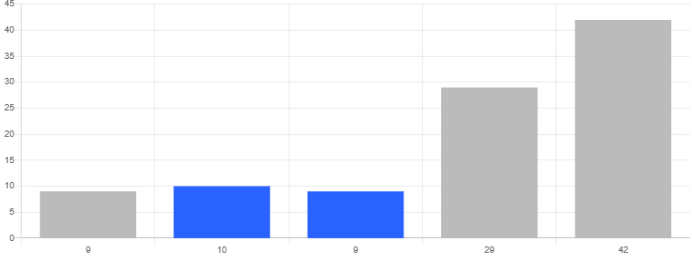
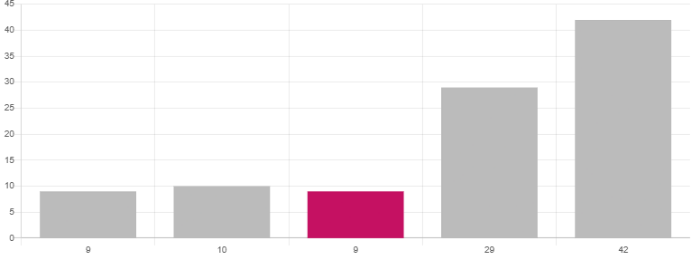
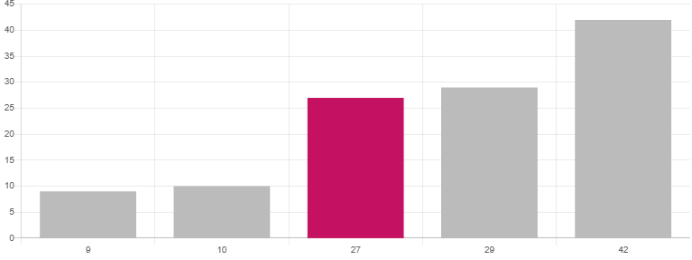
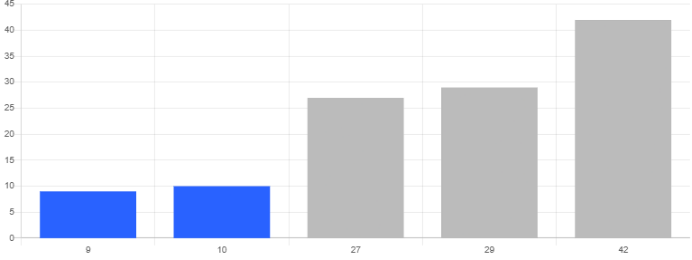
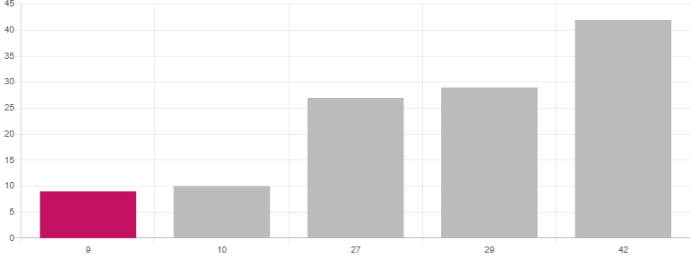
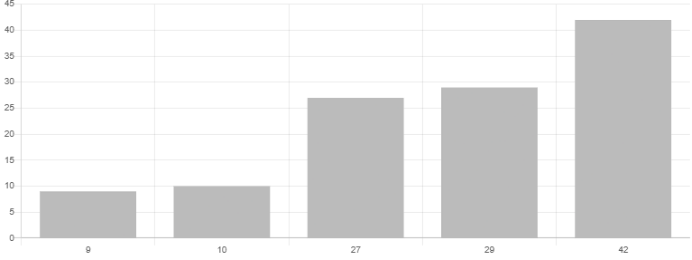
                         

Link para revisar el Sort animado: <https://algorithm-visualizer.org/brute-force/selection-sort>

Codigo Ejecutado:

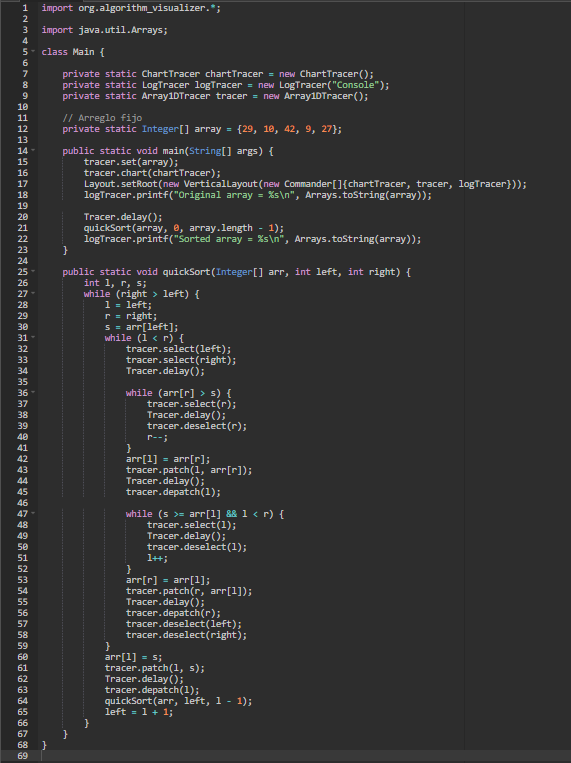


* + 1. **Quick Sort.**

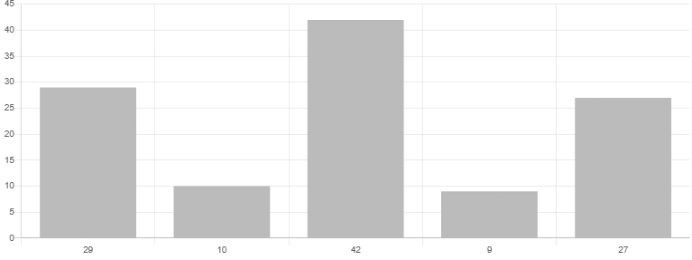
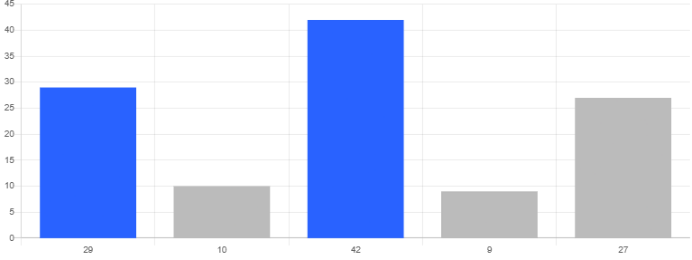
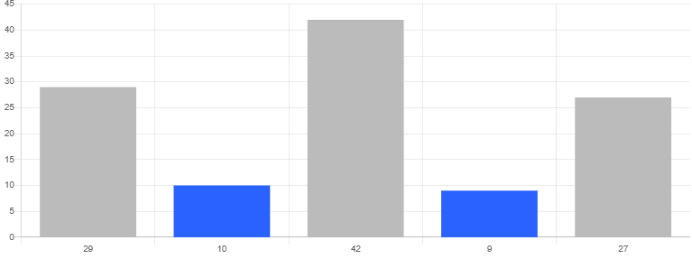
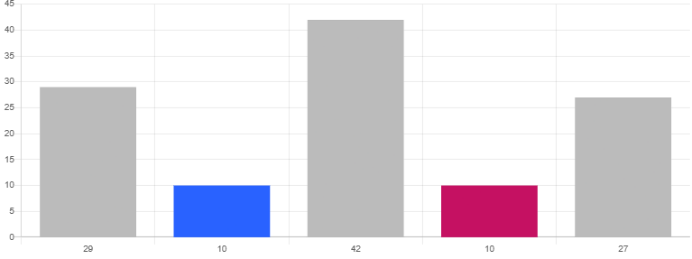
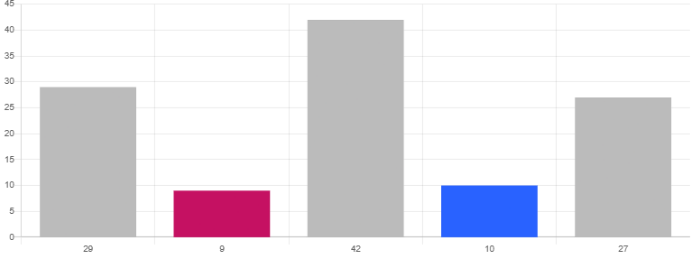
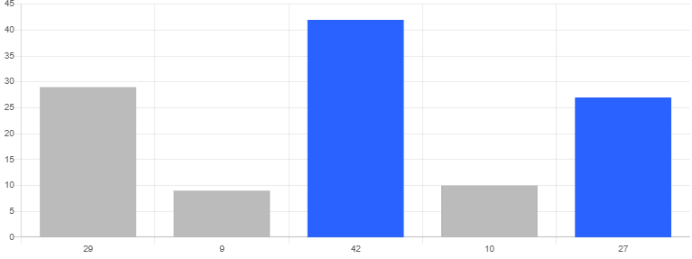
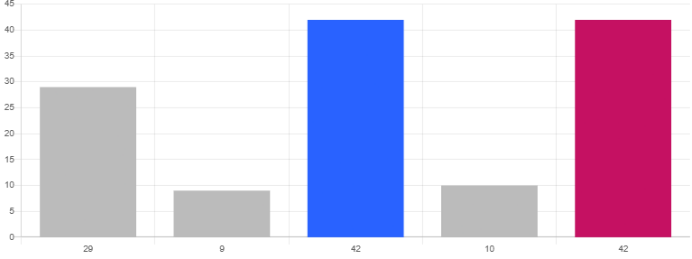
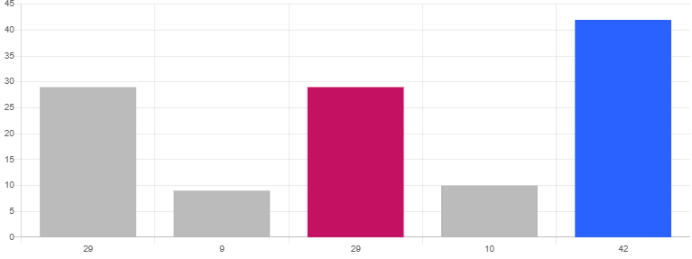
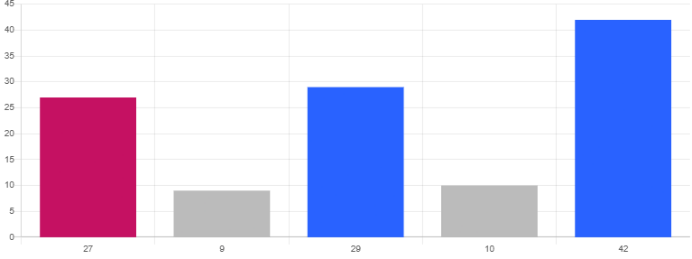
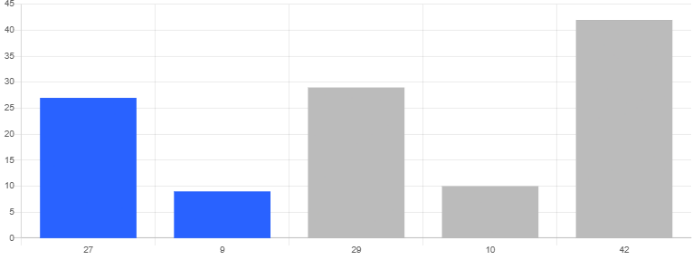
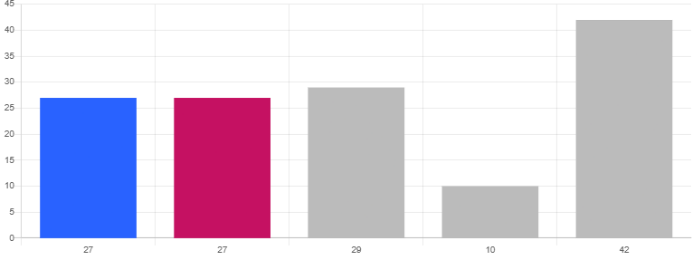
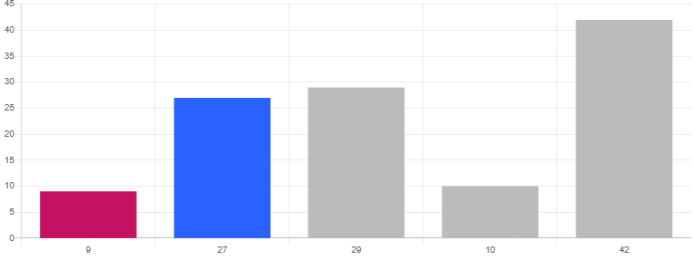
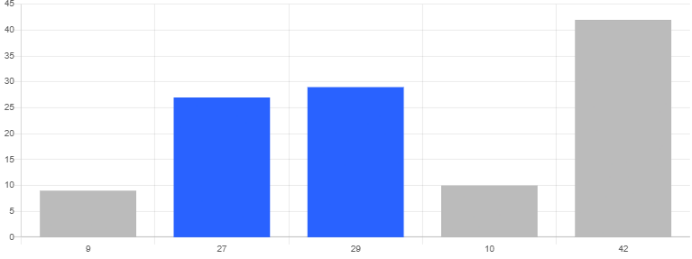
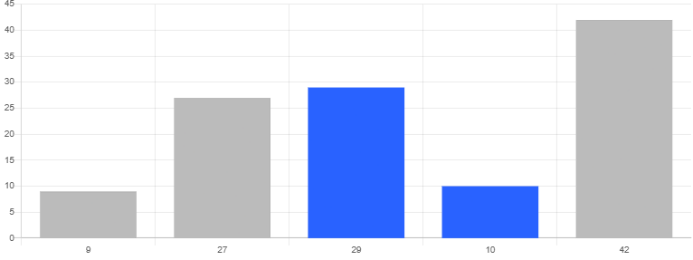
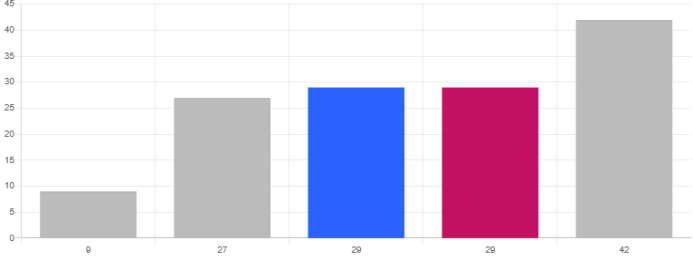
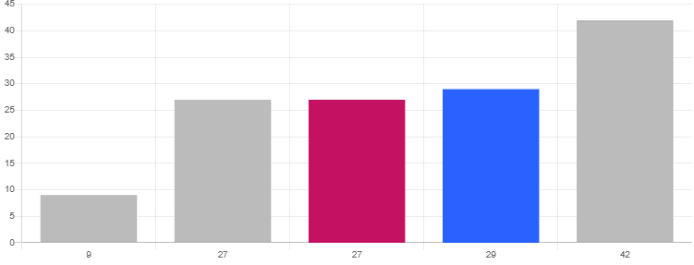
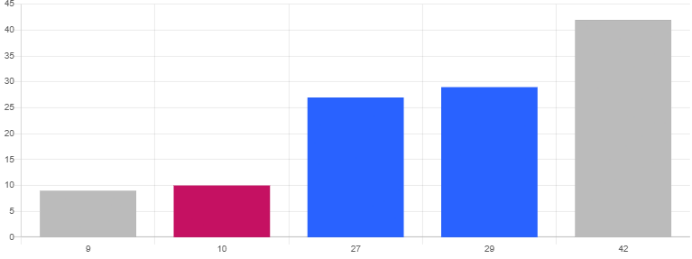
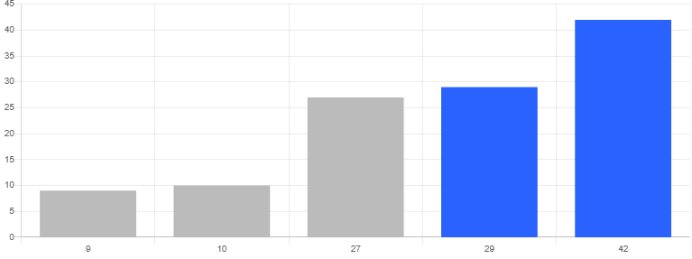
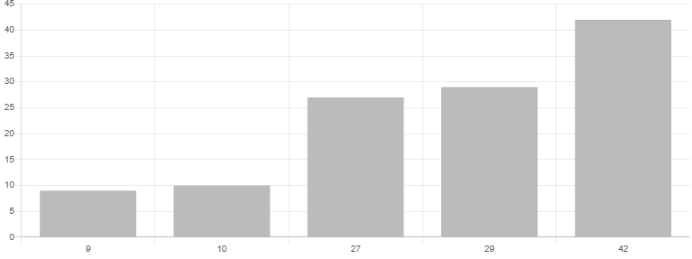
                   

Link para revisar el Sort animado: <https://algorithm-visualizer.org/divide-and-conquer/quicksort>

Codigo Ejecutado:

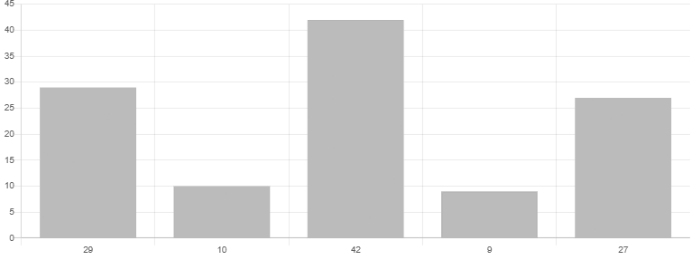
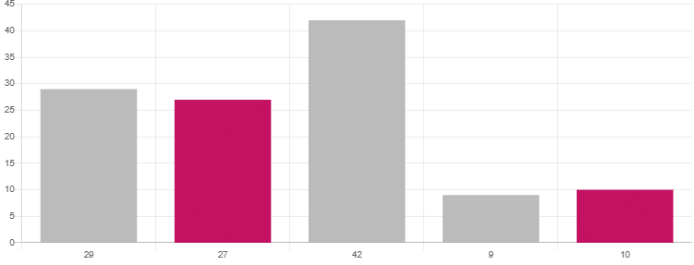
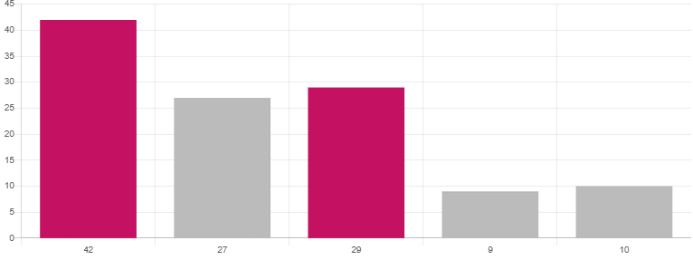
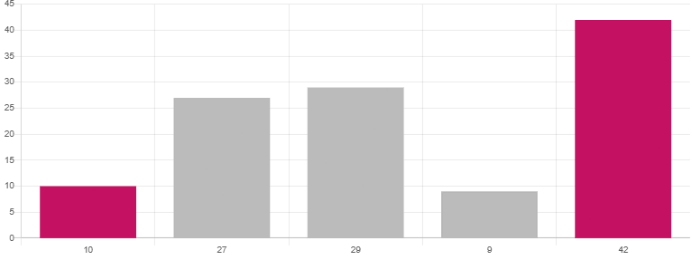
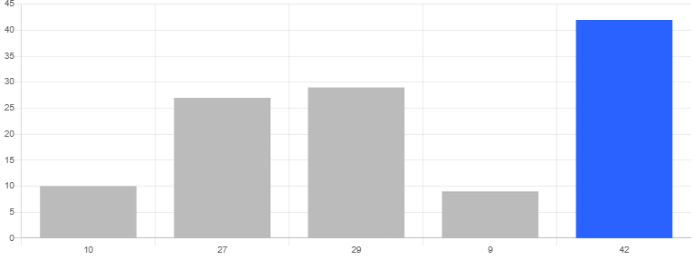
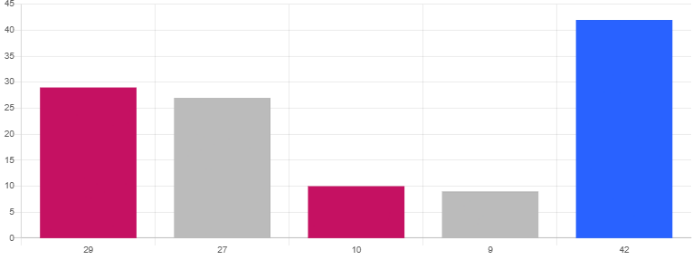
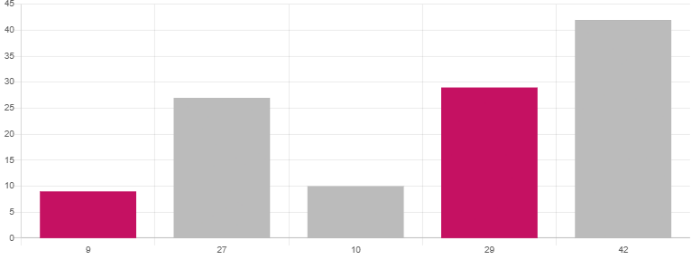
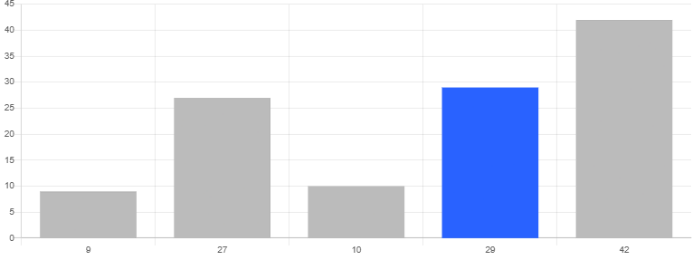
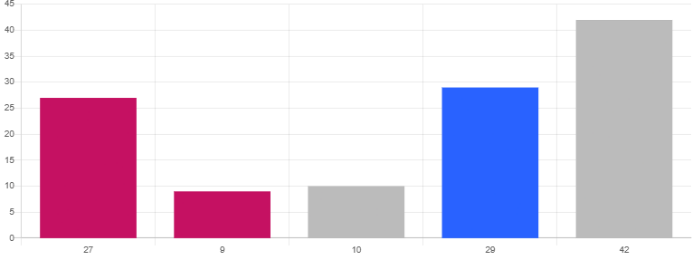
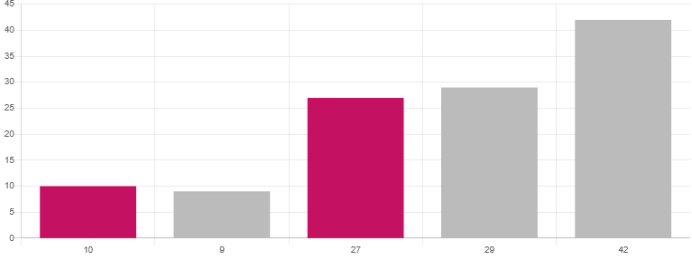
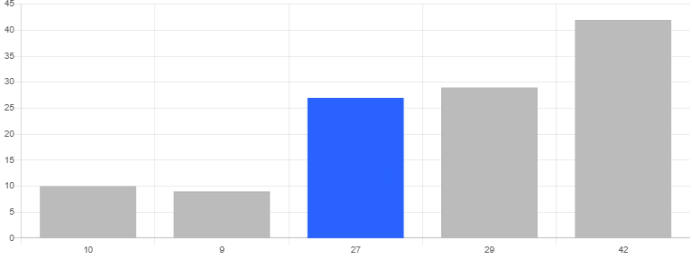
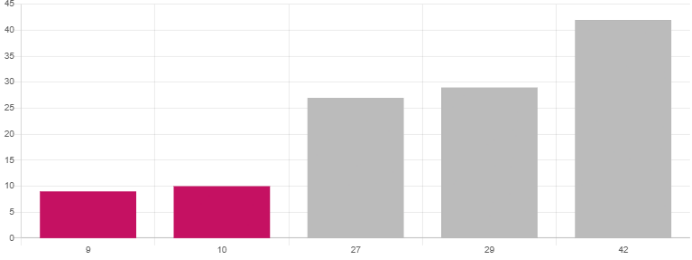
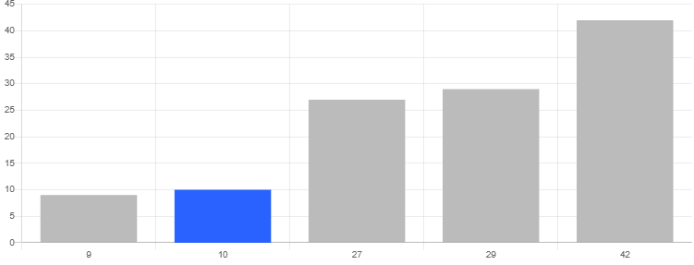
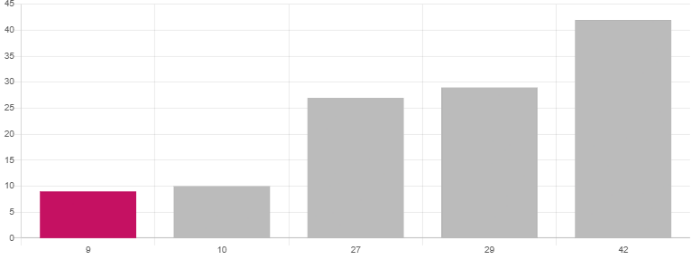
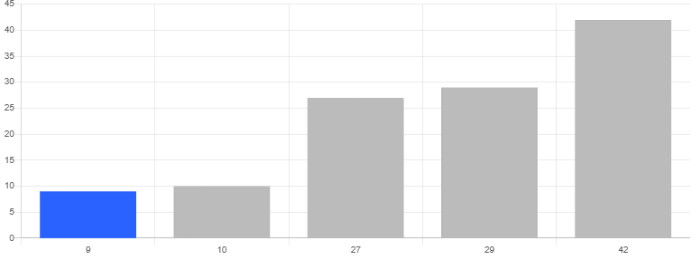
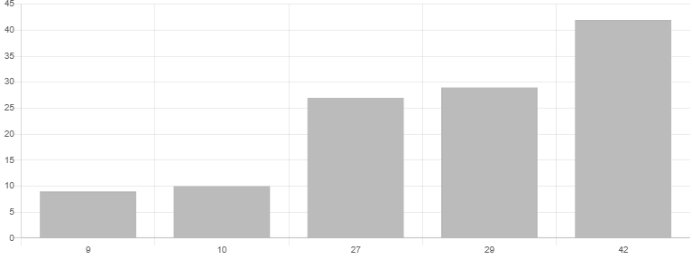


* + 1. **Shell Sort.**

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

* + 1. **Heap Sort.**

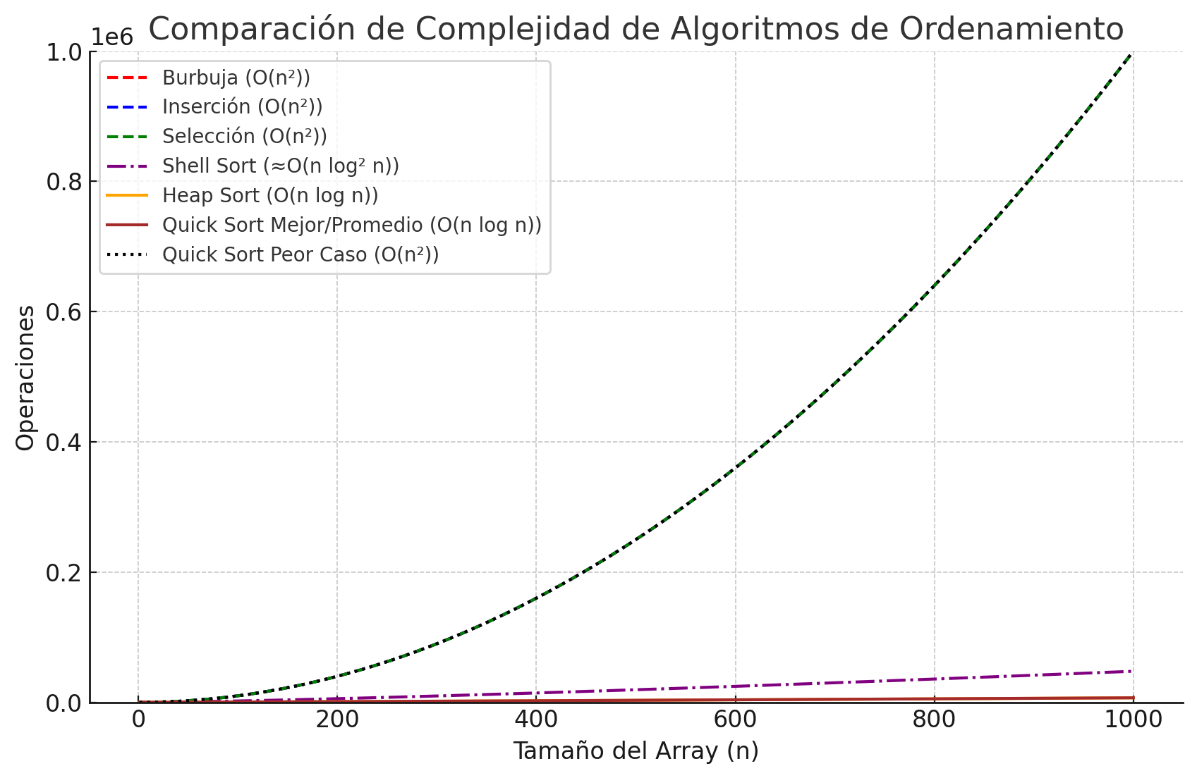
               

Link para revisar el Sort animado: <https://algorithm-visualizer.org/brute-force/heapsort>

* 1. Explicación grafica de la complejidad algorítmica.

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmo | Mejor Caso | Caso Promedio | Peor Caso |
| Burbuja |  |  |  |
| Inserción |  |  |  |
| Selección |  |  |  |
| Shell Sort |  | Depende del gap o |  |
| Heap Sort |  |  |  |
| Quick Sort |  |  |  |

|  |  |
| --- | --- |
| Expresiones | Explicación |
|  | * 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] |
|  | * El tiempo de ejecución crece de manera proporcional a 𝑛 * Ejemplo: Recorrer un array una vez. |
|  | * Si el tamaño de la entrada se duplica, el tiempo de ejecución se cuadruplica. * Ocurre cuando se usan dos ciclos anidados. |
|  | * Ocurre en algoritmos de "divide y vencerás" como Merge Sort y Quick Sort (en el mejor y promedio caso). * Ejemplo: Quick Sort, Heap Sort. |
|  | * Se da cuando el tamaño del problema se reduce a la mitad en cada paso. * Ejemplo: Búsqueda binaria en un array ordenado. |



* + 1. **Burbuja (Bubble Sort)**
* **Mejor Caso:** → Cuando el arreglo ya está ordenado, solo hace una pasada sin intercambios.
* **Caso Promedio y Peor Caso:** → En el peor de los casos (arreglo en orden inverso), tiene que hacer muchas comparaciones e intercambios.

**Ineficiente para grandes volúmenes de datos.**

* + 1. **Inserción (Insertion Sort)**
* **Mejor Caso:** → Si el arreglo ya está ordenado, solo revisa cada elemento una vez.
* **Caso Promedio y Peor Caso:** → Cuando el arreglo está desordenado, tiene que comparar e insertar cada elemento en su posición correcta.

**Bueno para listas pequeñas o casi ordenadas.**

* + 1. **Selección (Selection Sort)**
* **Siempre:** → → 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. **Shell Sort**
* **Mejor Caso:** → Depende del tamaño del "gap" usado.
* **Caso Promedio: Depende del gap**  o → No tiene una complejidad promedio bien definida.
* **Peor Caso:**  → 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. **Heap Sort**
* **Siempre:** → 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. **Quick Sort**
* **Mejor Caso y Caso Promedio:** → → Si se elige bien el pivote, el problema se divide en dos partes equilibradas.
* **Peor Caso:**  → 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.**