

Practica1: Analisis Experimental

Alumno: Jose Luis Obiang Ela Nanguang

Profesor: Luis Vicente Calderita Estévez

Asignatura: Análisis y Diseño de Algoritmos (ADA)

Fecha Entrega: 19/03/2024 23:59

Índice

Análisis

Enunciado

Objetivos

Desarrollo

Código

Github

Ejecución

Análisis del comportamiento temporal.

Conclusión

Herramientas

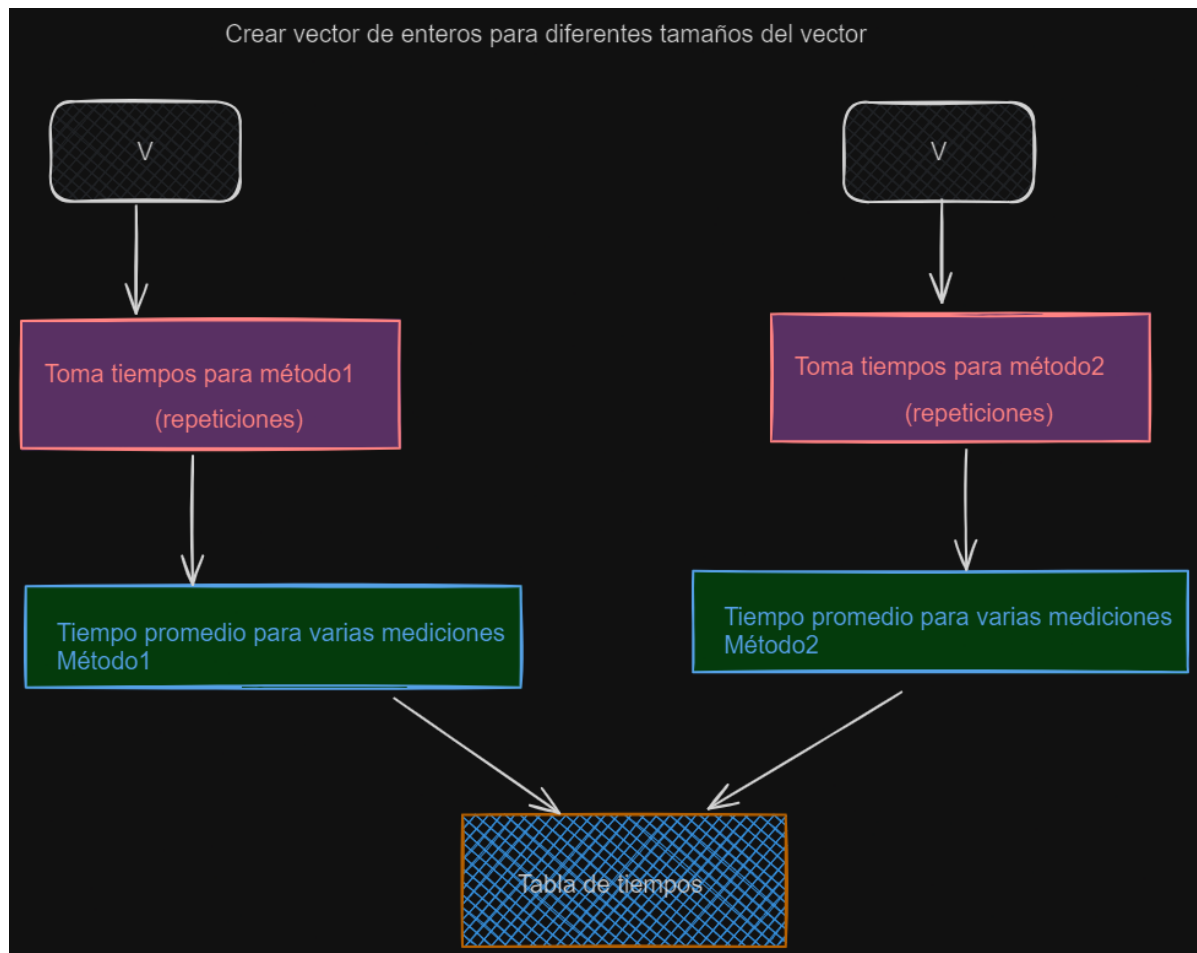
Análisis

Enunciado

1. Analizar el comportamiento temporal del algoritmo ante diferentes configuraciones de datos de entrada:
 - a. Caso Mejor.
 - b. Caso Peor.
 - c. Datos de entrada aleatorios.
2. Se deben realizar pruebas temporales con tamaños de problema (talla) crecientes.
3. Realizar varias mediciones para cada talla, para reducir el efecto de las perturbaciones en el sistema en la toma de tiempos.

Objetivos

- Estudiar el coste temporal real de algunos algoritmos de ordenación.
- Los algoritmos ordenarán ascendentemente un vector de enteros de diferente tamaño.
- Implementar los algoritmos de ordenación: Selection sort y Bubble sort.
- Comparar el coste entre los algoritmos implementados y los algoritmos ya disponibles (Cocktail Sort y Quicksort).



Desarrollo

Código

▼ Clases

[Main.java:](#)

```

1 package es.uex.cum.ada;
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.util.Arrays;
6
7 /**
8  * Práctica 1: Análisis experimental de la Eficiencia de algunos Algoritmos de Ordenación
9  * <ul>
10  * <li>Estudiar el coste temporal real de algunos algoritmos de ordenación</li>
11  * <li>Los algoritmos ordenaran ascendentemente un vector de enteros de diferente tamaño</li>
12  * <li>Implementar los algoritmos de ordenación Selection sort y Bubble sort</li>
13  * <li>Comparar el coste entre los algoritmos implementados y los algoritmos ya disponibles(Cocktail Sort y Bubble Sort)</li>
14  * <li>Analizar el comportamiento temporal del algoritmo ante diferentes configuraciones de datos de entrada
15  * <li>Caso Mejor</li>
16  * <li>Caso Peor</li>
17  * <li>Datos de entrada aleatorios</li>
18  * </ul>
19  * <li>Se deben realizar pruebas temporales con tamaños de problema (talla) crecientes.</li>
20  * <li>Realizar varias mediciones para cada talla para reducir el efecto de perturbaciones en el sistema en la toma de tiempos.</li>
21  * </ul>
22  *
23  * @author José Luis Obiang Ela Nanguang
24  * @version 1.0 2024-03-18
25  * @link JFrame, JPanel, JTextArea, JTable, JScrollPane, GridLayout, BorderLayout, Algoritmos,
26  * InterfazAlgoritmos
27  */
28
29 public class Main {
30
31     public static void main(String[] args) {
32         // Crear una instancia de Algoritmos
33         Algoritmos al = new Algoritmos();
34
35         // Definir los tamaños de los arrays a ordenar
36         int[] tallas = {2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000};
37
38         // Definir el número de repeticiones por cada tamaño
39         int repeticiones = 10;
40
41         // Definir los casos
42         String[] casos = {"Caso Mejor", "Caso Peor", "Caso Aleatorio"};
43
44         // Array para almacenar los casos, las tallas y tiempos medios de ejecución de cada algoritmo
45         String[][] tiempos = new String[casos.length][tallas.length][al.algoritmos.length];
46
47         // Iterar sobre los casos
48         for (int i = 0; i < casos.length; i++) {
49             JPanel casoPanel = new JPanel(new BorderLayout());
50             String caso = casos[i];
51             System.out.println("===== " + caso + " =====");
52
53             // Iterar sobre los tamaños de los arrays
54             for (int j = 0; j < tallas.length; j++) {
55                 // Crear una ventana Swing
56                 JFrame frame = new JFrame("Arrays - " + caso + " - Talla: " + tallas[j]);
57                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
58                 // Crear un panel principal para la ventana
59                 JPanel panel = new JPanel(new GridLayout(2, 1));
60                 JTextArea originalTextArea = new JTextArea();
61                 JTextArea ordenadoTextArea = new JTextArea();
62
63                 int talla = tallas[j];
64                 long[] tiemposTotales = new long[al.algoritmos.length];
65                 System.out.println("Talla del array: " + talla);
66
67                 // Generar el array según el caso
68                 int[] arrayOriginal = al.generarArray(talla, caso);
69                 // Mostrar el array original (solo para comprobar que funciona)
70                 System.out.println("Array original: " + Arrays.toString(arrayOriginal));
71
72                 int[] arrayOrdenado = new int[talla];
73                 for (int l = 0; l < al.algoritmos.length; l++) {
74                     // Medir el tiempo de ejecución de cada algoritmo de ordenación
75                     long inicio = System.currentTimeMillis();
76                     for (int k = 0; k < repeticiones; k++) {
77                         // Ordenar el array con el algoritmo correspondiente
78                         arrayOrdenado = switch (l) {
79                             case 0 -> al.bubbleSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length));
80                             case 1 -> al.selectionSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length));
81                             case 2 -> al.quickSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length), 0, arrayOriginal.length - 1);
82                             case 3 -> al.cocktailSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length));
83                             default -> arrayOriginal;
84                         };
85
86                         long fin = System.currentTimeMillis();
87                         tiemposTotales[l] += (fin - inicio);
88                     }
89                 }
90
91                 // Calcular los tiempos medios
92                 for (int l = 0; l < al.algoritmos.length; l++) {
93                     tiempos[i][j][l] = String.valueOf(tiemposTotales[l] / (double) repeticiones);
94                     // Mostrar el array ordenado con el algoritmo correspondiente (solo para comprobar que funciona)
95                     System.out.println("Array ordenado con algoritmo " + al.algoritmos[l] + ": " + Arrays.toString(arrayOrdenado));
96                 }
97
98                 // Mostrar el array original y el array ordenado
99                 originalTextArea.setEditable(false);
100                 originalTextArea.append("Array Original (" + caso + " - Talla: " + talla + "):\n" + Arrays.toString(arrayOriginal));
101                 ordenadoTextArea.setEditable(false);
102                 ordenadoTextArea.append("Array Ordenado (" + caso + " - Talla: " + talla + "):\n" + Arrays.toString(arrayOrdenado));
103
104                 panel.add(originalTextArea);
105                 panel.add(ordenadoTextArea);
106
107                 frame.add(panel);
108                 frame.setSize(400, 300);
109                 frame.setLocationRelativeTo(null); // Centrar la ventana en la pantalla
110                 frame.setVisible(true);
111
112             }
113
114         }
115
116         System.out.println("===== \n");
117
118         // Mostrar los resultados finales en una tabla
119         al.mostrarResultados(casos, tallas, tiempos);
120
121         JFrame frameFinal = new JFrame("Análisis experimental");
122         frameFinal.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
123         // Crear un panel principal para la ventana final
124         JPanel panelFinal = new JPanel(new GridLayout(1, casos.length));
125
126         // Crear y agregar tablas para cada caso
127         for (int i = 0; i < casos.length; i++) {
128             String caso = casos[i];
129             JTable table = al.createTable(caso, tallas, tiempos[i]);
130             JScrollPane scrollPane = new JScrollPane(table);
131             panelFinal.add(scrollPane);
132         }
133
134         // Agregar el panel principal a la ventana
135         frameFinal.add(panelFinal);
136
137         // Configurar la ventana y hacerla visible
138         frameFinal.setSize(800, 600);
139         frameFinal.setLocationRelativeTo(null); // Centrar la ventana en la pantalla
140         frameFinal.setVisible(true);
141
142     }
143 }
144
145 }
146

```

InterfazAlgoritmos.java

```
1 package es.uex.cum.ada;
2
3 import javax.swing.*;
4 import javax.swing.table.DefaultTableCellRenderer;
5 import javax.swing.table.DefaultTableModel;
6 import java.awt.*;
7 import java.util.Arrays;
8 import java.util.Random;
9
10 /**
11  * Interfaz que contiene todas las funciones a usar en el programa
12  * @author José Luis Obiang Ela Nanguang
13  * @version 1.0 2024-03-18
14  */
15
16 public interface InterfazAlgoritmos {
17     int[] generarArray(int talla, String caso);
18     void mostrarResultados(String[] casos, int[] tallas, String[][][] tiempos);
19     int[] bubbleSort(int arr[]);
20     int[] selectionSort(int arr[]);
21     int[] cocktailSort(int arr[]);
22     int[] quickSort(int a[], int izq, int der);
23     void intercambiar(int a[], int ind1, int ind2);
24     int mediana3(int a[], int izq, int der);
25     int[] generateRandomArray(int size);
26     int[] generateSortedArray(int size);
27     int[] generateReverseSortedArray(int size);
28     JTable createTable(String caso, int[] tallas, String[][] tiempos);
29
30 }
31
32
```

Algoritmos.java

[illegible]

Github

<https://github.com/Jloen1999/ADA>

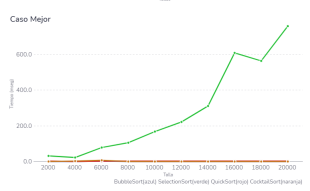
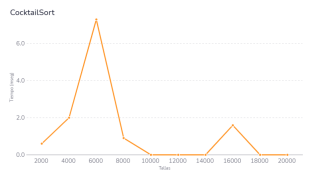
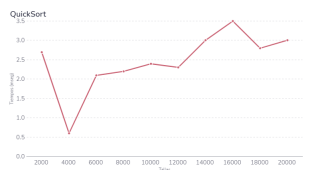
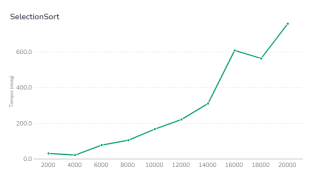
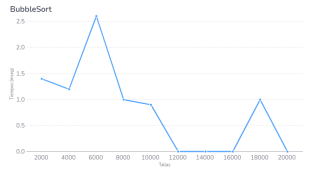
Ejecución

Análisis experimental															
Caso	Algoritmo	bubblesort	selectionsort	quicksort	cocktailsort	Caso	Algoritmo	bubblesort	selectionsort	quicksort	cocktailsort	Caso	Algoritmo	bubblesort	selectionsort
2000	Mejor	1.4	32.7	2.7	0.6	2000	Peor	41.8	46.8	11.3	64.2	2000	Mejor	28.3	47.4
4000		1.2	23.9	0.6	2.0	4000		119.5	180.0	0.5	71.3	4000		172.4	225.1
6000		2.6	79.6	2.1	7.3	6000		232.6	320.5	1.2	96.1	6000		368.9	359.6
8000		1.0	106.0	2.2	0.9	8000		456.0	556.4	1.7	184.9	8000		922.1	610.9
10000		0.9	168.1	2.4	0.0	10000		941.9	942.0	4.5	429.6	10000		1194.6	843.9
12000		0.0	222.6	2.3	0.0	12000		1277.8	1210.5	2.9	423.7	12000		1417.3	1514.6
14000		0.0	311.0	3.0	0.0	14000		1197.2	1639.5	2.2	550.2	14000		2355.8	2293.4
16000		0.0	607.4	3.5	1.6	16000		1391.4	2131.4	2.8	725.0	16000		2722.6	2108.1
18000		1.0	563.8	2.8	0.0	18000		1948.1	3020.6	6.1	1487.9	18000		3147.5	2788.4
20000		0.0	757.6	3.0	0.0	20000		1999.9	3577.6	4.1	1281.4	20000		3716.0	3490.9

Análisis del comportamiento temporal.

▼ Caso Mejor

- En el caso mejor, donde el array ya está ordenado, se observa que los algoritmos de ordenación **Bubble Sort** y **Cocktail Sort** son los más eficientes en términos de tiempo.
- **Bubble Sort** y **Cocktail Sort** tienen tiempos muy bajos en comparación con los otros algoritmos en este caso. Y por lo que se observa el tiempo de ordenación es inversamente proporcional al tamaño del array, es decir, el tiempo de ordenación se achica cuanto mayor es el tamaño.
- **Selection Sort** muestra tiempos más altos en comparación con los otros algoritmos. El tiempo de ordenación aumenta conforme aumenta el tamaño del array.
- **Quick Sort** también muestra tiempos bajos, aunque ligeramente más altos que **Bubble Sort** y **Cocktail Sort**.
- En general, los tiempos son relativamente bajos y aumentan gradualmente a medida que aumenta el tamaño del arreglo, pero los algoritmos más eficientes (**Bubble Sort**, **Cocktail Sort** y **Quick Sort** respectivamente) mantienen sus ventajas.

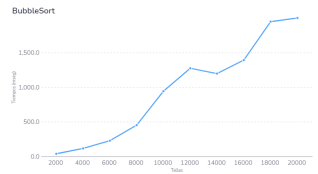


▼ Caso Peor

- En el caso peor, donde el arreglo está invertido, se observa que los tiempos de ejecución de los algoritmos son significativamente más altos en comparación con el caso mejor.
- **Bubble Sort** y **Selection Sort** tienen tiempos muy altos. **Selection Sort** muestra el peor rendimiento entre los cuatro algoritmos con valores gradualmente mayores que en el caso mejor.
- Los tiempos de ejecución aumentan drásticamente a medida que aumenta el tamaño del arreglo, con **Quick Sort** muestra la menor tasa de crecimiento entre los algoritmos. Sigue siendo rápido. Seguido de él tenemos el **Cocktail Sort**, **Bubble Sort** y por último **Selection Sort**(el más lento en este caso).

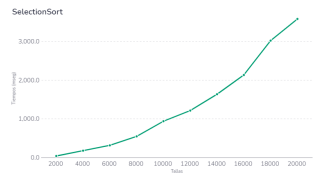
BubbleSort

2000	41.8
4000	118.5
6000	232.6
8000	456.0
10000	541.5
12000	1,277.8
14000	1,197.2
16000	1,391.4
18000	1,948.1
20000	1,999.9



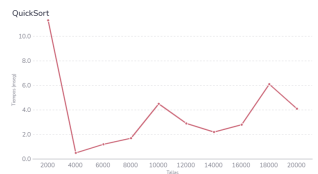
SelectionSort

2000	46.8
4000	180.0
6000	320.5
8000	556.4
10000	942.0
12000	1,210.5
14000	1,639.5
16000	2,131.4
18000	3,020.6
20000	3,577.6



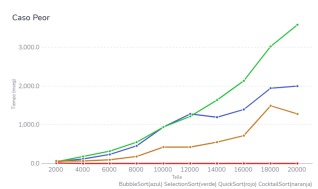
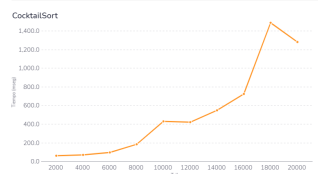
QuickSort

2000	11.3
4000	0.5
6000	1.2
8000	1.7
10000	4.5
12000	2.9
14000	2.2
16000	2.8
18000	6.1
20000	4.1



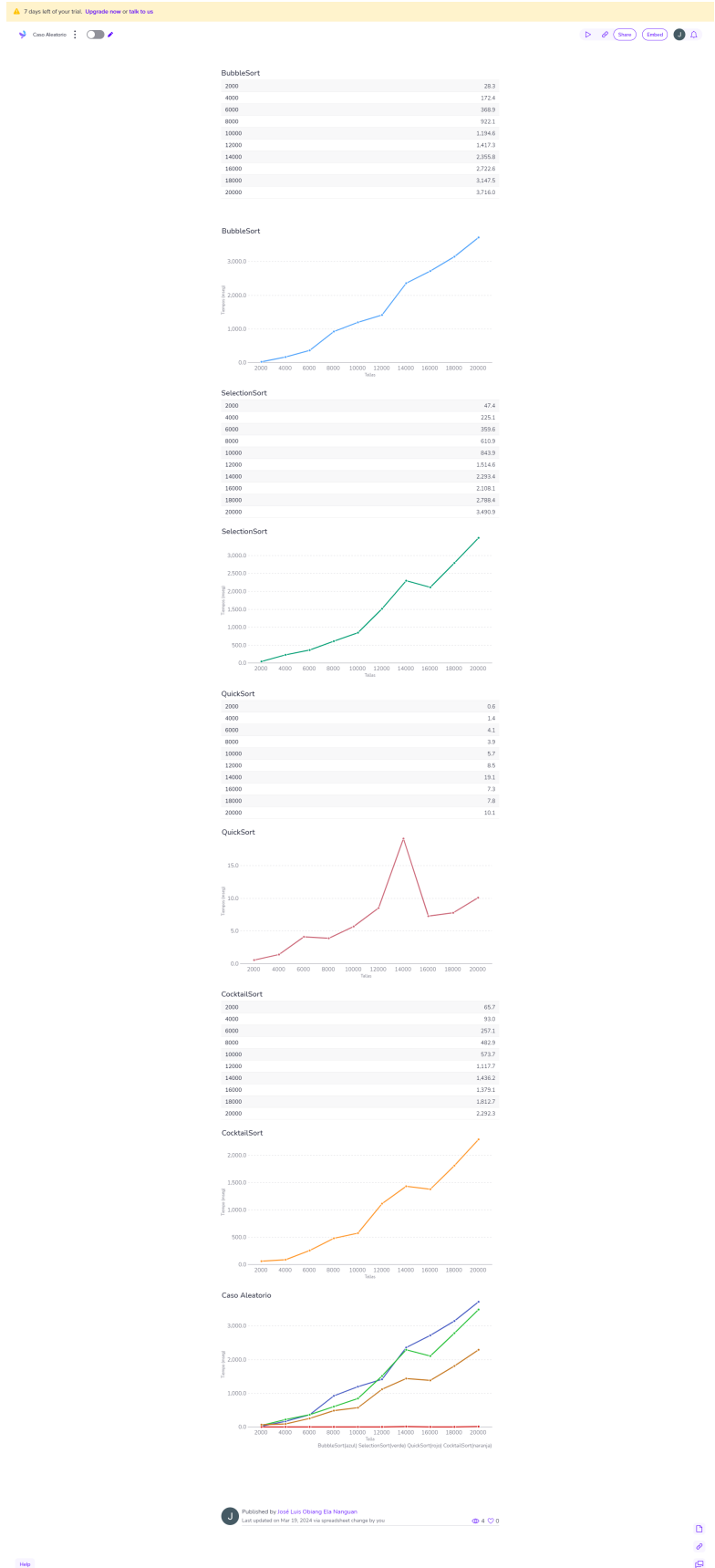
CocktailSort

2000	64.2
4000	71.3
6000	96.1
8000	184.9
10000	429.6
12000	423.7
14000	550.2
16000	970.0
18000	1,467.9
20000	1,281.4



▼ Caso Aleatorio

- En el caso aleatorio o medio, donde los datos de entrada son aleatorios, se observa un comportamiento intermedio en términos de tiempos de ejecución.
- **Quick Sort** sigue siendo el algoritmo más eficiente en este caso, seguido de **Selection Sort**.
- En este orden, tanto el **Bubble Sort** como el **Selection Sort** y **Cocktail Sort** muestran un rendimiento muy bajo especialmente a medida que aumenta el tamaño del arreglo.
- Los tiempos de ejecución de todos los algoritmos aumentan de manera significativa a medida que aumenta el tamaño del array, con **Quick Sort** mostrando la menor tasa de crecimiento.
- **Bubble Sort y Selection Sort** muestran un comportamiento similar en términos de tiempos de ejecución en este caso, aunque **Bubble Sort** tiende a ser un poco más lento.



Conclusión

Al finalizar el análisis temporal del coste de ejecución de los diferentes algoritmos de ordenación mencionados he podido llegar a la conclusión de que el **Quick Sort** muestra consistentemente el mejor rendimiento en todos los casos. El **Bubble Sort** en comparación a los demás algoritmos tiene un rendimiento pésimo tanto en el caso peor como en aleatorio seguido del **Selection Sort** y **Cocktail Sort** respectivamente, mientras que en el mejor caso tiene un rendimiento similar al **Quick Sort** y **Cocktail Sort**.

Selection Sort en el mejor caso es el más lento de todos.

Herramientas

- Gráficas: [grid](#)
- Diagrama: [excalidraw](#)
- Code Snapshot: [carbon.now.sh](#)