

# Resumen Algoritmos de Sorting

Por: Ramiro Seltzer

Algoritmo	Complejidad Temporal (Peor Caso)	Complejidad Temporal (Caso Promedio)	Estabilidad	Notas
Bubble Sort	$O(n^2)$	$O(n^2)$	Estable	Simple, ineficiente para mucha información - <i>Comparación</i>
Selection Sort	$O(n^2)$	$O(n^2)$	Inestable	Simple, ineficiente para mucha información - <i>Comparación</i>
Insertion Sort	$O(n^2)$	$O(n^2)$	Estable	Eficiente para pequeños datos, adaptativo - <i>Comparación</i>
Merge Sort	$O(n \log n)$	$O(n \log n)$	Estable	Eficiente para grandes datos, ocupa mucho espacio - <i>Comparación</i>
Quick Sort	$O(n^2)$ (peor caso)	$O(n \log n)$	Inestable	Eficiente para grandes datos - <i>Comparación</i>
Heap Sort	$O(n \log n)$	$O(n \log n)$	Inestable	Eficiente para grandes datos, no adaptativo
Radix Sort	$O(nk)$	$O(nk)$	Estable	Solo para números enteros, complejidad lineal
Counting Sort	$O(n + k)$	$O(n + k)$	Estable	Solo para números enteros acotados, lineal
Bucket Sort	$O(n^2)$ (avg case)	$O(n + k)$	Estable	Útil cuando la distribución es uniforme - <i>Complejidad Temporal Promedio</i>

**Algoritmo Estable:** Un algoritmo se dice estable cuando tenemos elementos cuyo valor es el mismo (en posiciones  $i, j$ ) tales que, si se aplica un algoritmo de sorting, la posición relativa entre ambos elementos se preserva.

Ej:  $\rightarrow [4_0, 2, 1, 3, 4_1, 5] \Rightarrow [1, 2, 3, 4_0, 4_1, 5]$ .

**Algoritmo in-place:** Se dice que un algoritmo es *in-place* cuando no necesitamos inicializar un nuevo array *res* para almacenar el resultado de ordenar nuestro array de entrada.

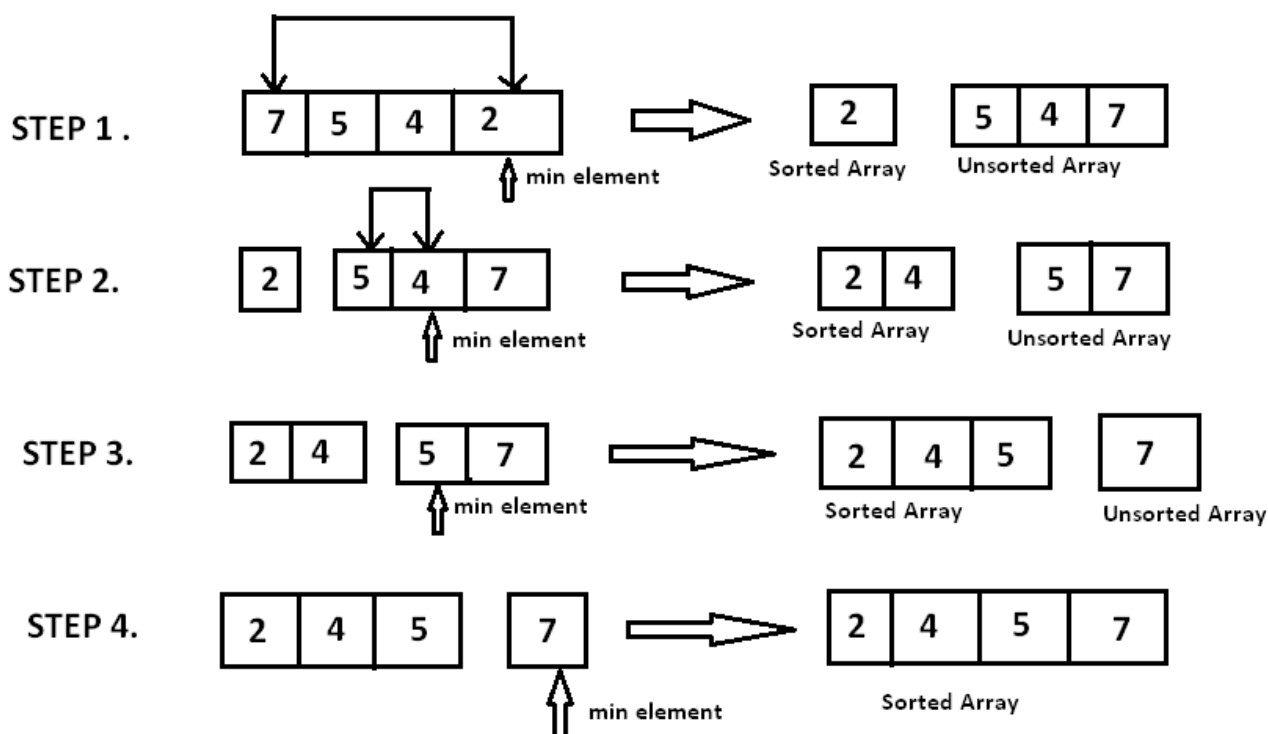
## Sorting por comparación

### Selection Sort

La idea detrás de Selection Sort es:

- Dado un array de enteros  $a$ , buscamos el elemento máximo de la lista y el mínimo.
- Una vez que los encontramos. Los intercambiamos de posición.
- El primer elemento de nuestra lista ya se encuentra ordenado. Lo separamos. Buscamos el siguiente mínimo.
- Repetimos el proceso hasta que tengamos un solo elemento  $\Rightarrow$  todos los demás elementos ya fueron ordenados.

$$Costo = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \mathcal{O}(n^2)$$

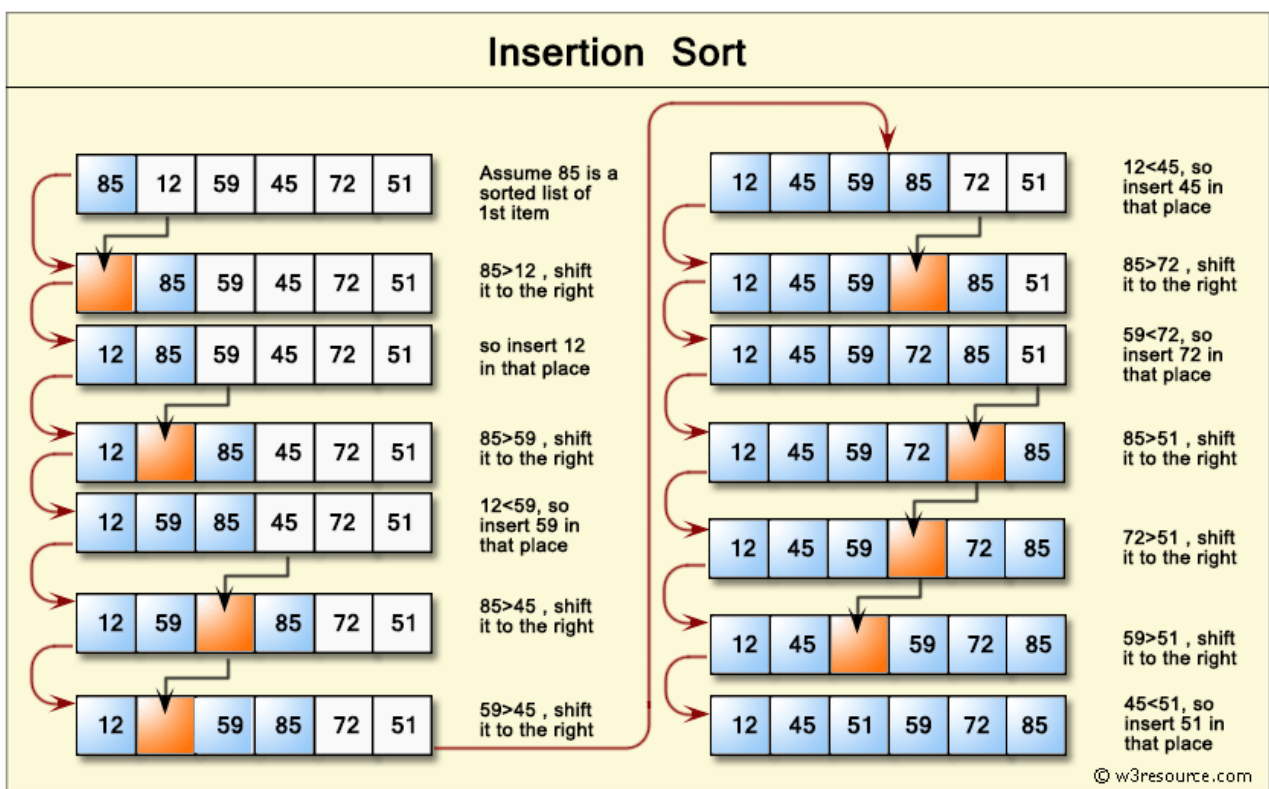


### Insertion Sort

La idea detrás de Insertion Sort es la siguiente:

- Iniciamos con el primer elemento. Al recibir una segunda carta, evaluamos si el primer elemento es menor que el que hemos recibido.
- En caso de que la nueva carta sea de mayor valor, desplazamos nuestra primera carta hacia la izquierda.
- Repetimos este proceso de manera iterativa hasta que logramos organizar nuestro conjunto de cartas en orden ascendente. Este método sencillo pero efectivo refleja la esencia del Insertion Sort.
- Excelente para conjuntos pequeños de información.

$$Costo = \sum_{i=1}^{n-1} \sum_{j=i-1}^0 1 = \mathcal{O}(n^2)$$



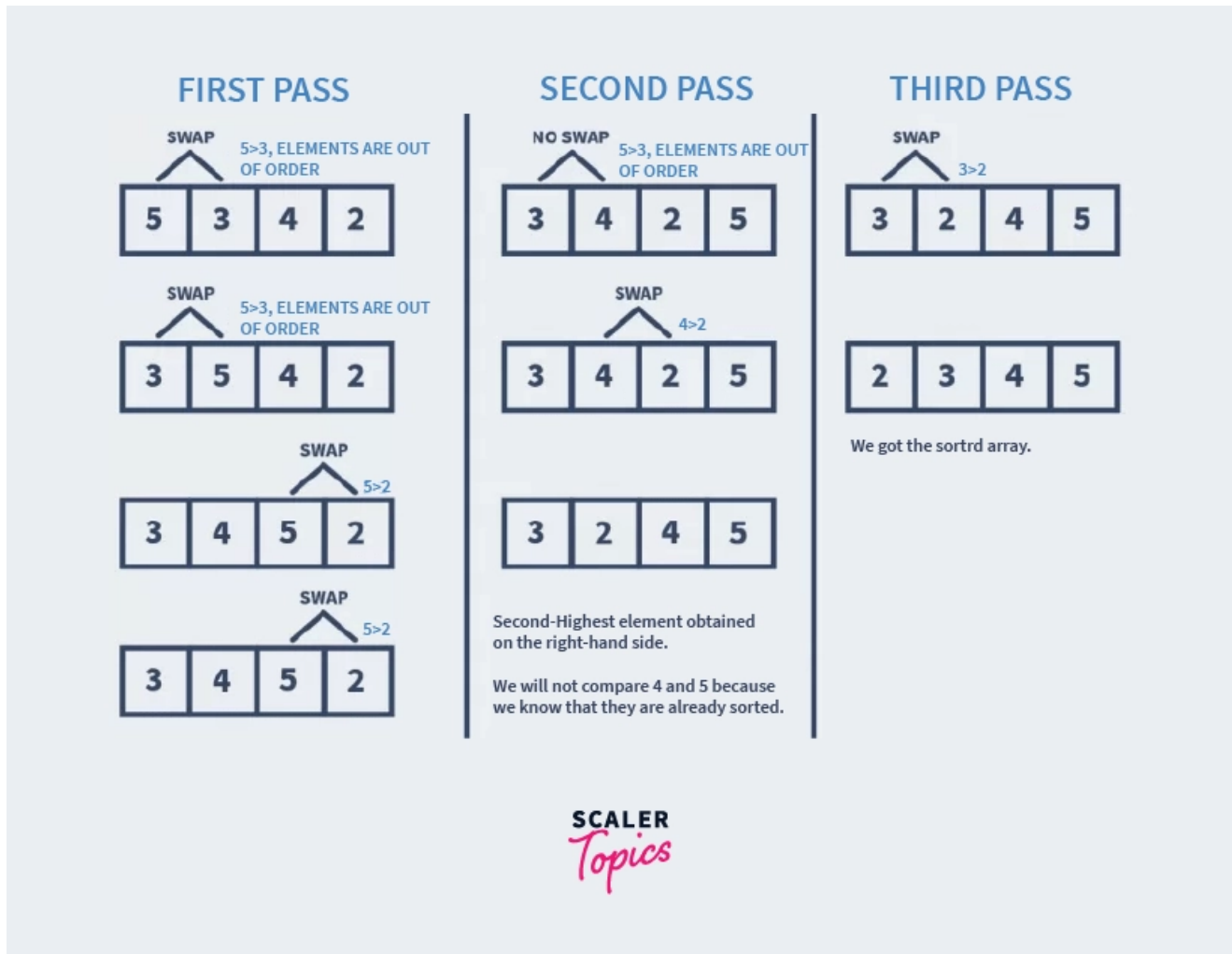
## Bubble Sort

El Algoritmo de Bubble Sort sigue una dinámica conceptualmente igual a Selection Sort. Este método se basa en la siguiente idea:

- Dado un arreglo de enteros, realizamos la comparación entre los dos primeros elementos. Si están desordenados, los ordenamos correctamente.

- Para todos los números del arreglo, realizamos esta comparación con el valor que se encuentra contiguo. Tendremos que realizar esta operación un total de  $n^2$  veces para ordenar el arreglo en su totalidad.
- Dentro de cada iteración del primer bucle, hay otro bucle que se ejecuta  $n - i - 1$  veces. Aquí,  $i$  representa el número de iteraciones completas del primer bucle. En la primera iteración, el segundo bucle se ejecuta  $n - 1$  veces, en la segunda iteración se ejecuta  $n - 2$  veces, y así sucesivamente.

$$\text{Costo} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} 1 = \mathcal{O}(n^2)$$

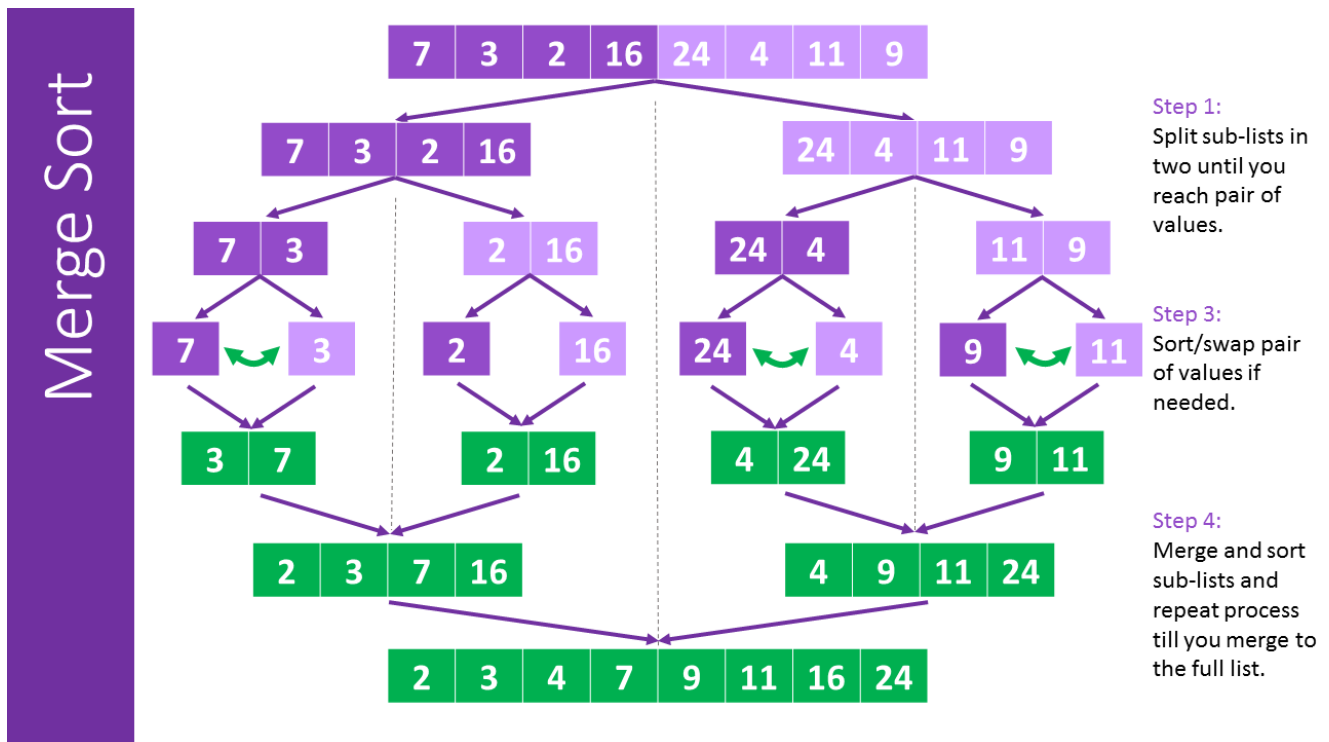


## Merge Sort

El Algoritmo de Merge Sort, concebido por el genio *John Von Neumann*, es una joya de Sorting Comparativo que se basa en la ingeniosa estrategia de *Divide & Conquer*. Este método, un ballet algorítmico, se desenvuelve de manera recursiva con la siguiente danza:

- Se le presenta un conjunto, representado por el array  $a$ , y lo divide con maestría en dos mitades. De forma consecutiva, invoca a sí mismo sobre cada una de estas mitades, proclamando la grandiosidad de  $mergeSort(a)$ .

- Cuando enfrenta el caso desafiante de  $n = 2$ , realiza un elegante dueto entre los dos elementos, orquestando su ordenación con gracia y precisión.
- Una vez que todos los elementos del conjunto, ahora apareados y ordenados, han ejecutado su magnífico baile individual, llega el momento culminante: el *merge*. Esta función magistralmente fusiona los elementos previamente separados, devolviendo así una sinfonía ordenada y completa, como la que se apreciaba al inicio de esta sublime composición.



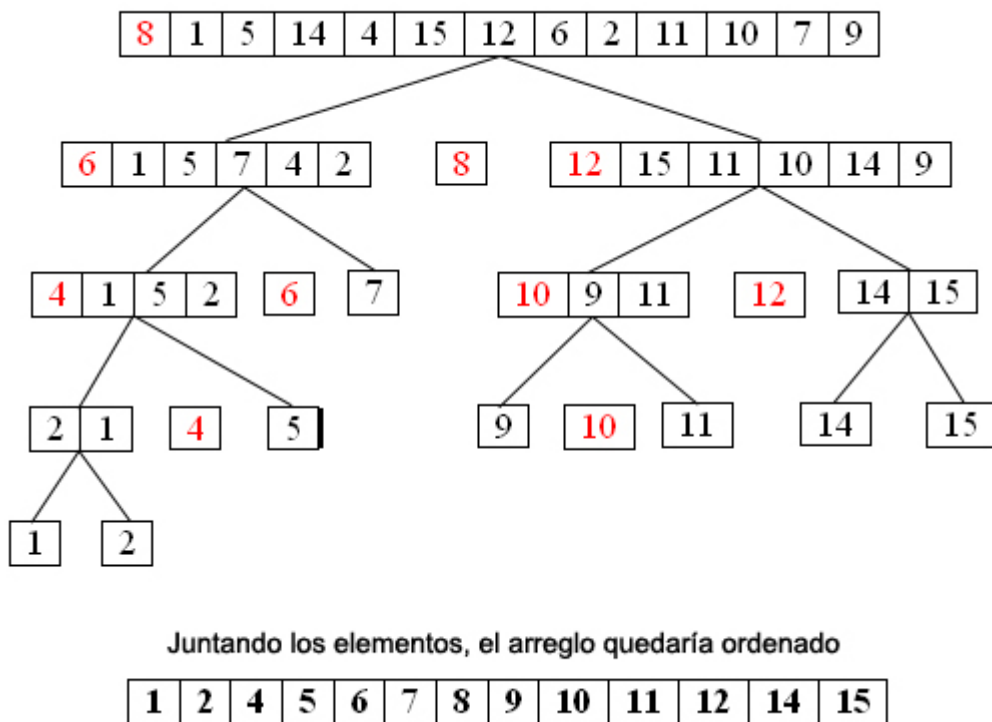
## Quick Sort

Quick Sort es un algoritmo de sorting por comparación inventado por *C.A.R Hoare* (tripas de Hoare) el cual se basa en elegir elementos de pivote y comparar cada número con el respectivo pivote.

La idea es la siguiente:

- Selecciona un elemento como pivote. Es importante tener una elección acertada del pivote (puede influir en la complejidad temporal)
- Particiona el array en dos subarrays: uno con elementos menores que el pivote y otro con elementos mayores.
- Aplica el mismo proceso de manera recursiva a los subarrays. Llama a QuickSort para los subarrays menores y mayores que el pivote.
- Cuando los subarrays son de tamaño 0 o 1, el array está ordenado. Los subarrays iguales al pivote ya están ordenados debido a la recursividad (por su caso base).

- Juntando todos los elementos, como es *in-place* nos queda el arreglo ordenado.  
=> Si

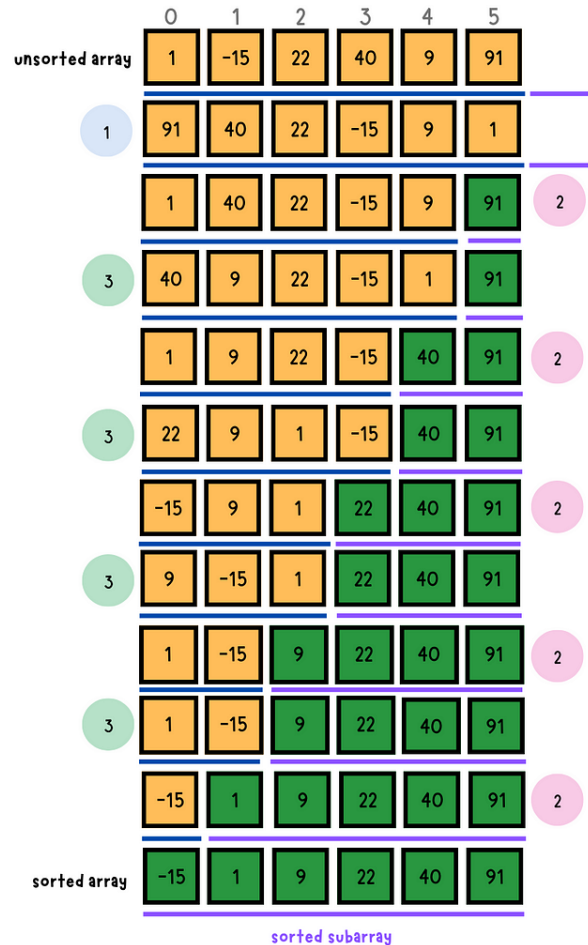
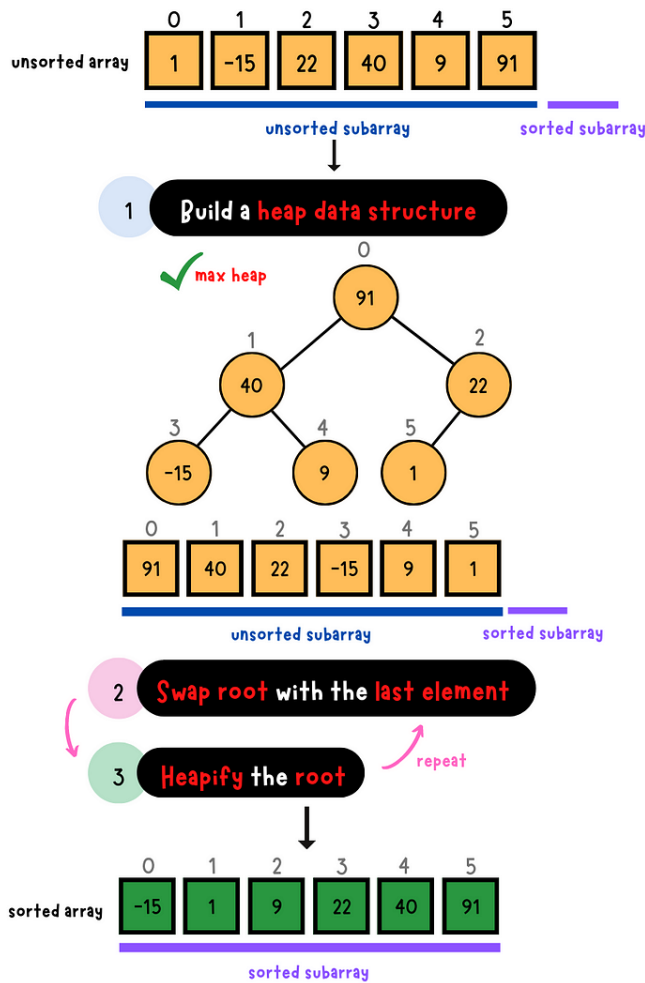



## Heap Sort

Heap Sort se basa en ordenar elementos utilizando como herramienta la estructura de datos *Max Heap*, la idea consiste en:

- Dado un array  $a$ , utilizamos el algoritmo de *heapify* para construir nuestro Max Heap.
- Una vez que tenemos nuestro arreglo de Max Heap, sabiendo que la raíz es el elemento máximo, lo insertamos en la última posición del arreglo.
- Puesto que ya lo eliminamos de mi árbol, nuevamente heapifico mi estructura sin tener en cuenta el último dato del arreglo.
- Repito hasta tener 1 solo elemento lo cual implica que nuestro arreglo ya está ordenado.

# Heap Sort Algorithm



 **time complexity:  $O(n \log(n))$**   $n$ : the total number of elements in the input array.

## Sorting por no comparación

### Counting Sort

La idea detrás de Counting Sort es poder ordenar un conjunto de números dentro de un array sabiendo que tenemos una cota para el valor máximo de un elemento del array. Sabiendo esto, el algoritmo plantea lo siguiente:

- Encontramos el valor máximo del arreglo  $a$  para inicializar un nuevo arreglo de tamaño  $\text{maximo}(a) + 1$  (inicializado en 0 todas las posiciones el cual contendrá la cantidad de repeticiones de cada uno de los números (donde el índice del arreglo representa el número)).
- Iteramos por cada elemento de  $a$  e incrementamos en 1 por cada aparición del elemento en mi arreglo *cuenta*.

- Una vez que tenemos todas las apariciones de los elementos de  $a$ , simplemente "expandimos" en mi arreglo que devuelvo de forma creciente el elemento con su cantidad de apariciones.

### Step 1:

	0	1	2	3	4	5	6	7	max
inputArray	2	5	3	0	2	3	0	3	5

Counting Sort



### Step 2:

	0	1	2	3	4	5
countArray	0	0	0	0	0	0

Counting Sort





### Step 3 :

	0	1	2	3	4	5
countArray	2	0	2	3	0	1

Counting Sort



### Step 5 :

	0	1	2	3	4	5	6	7
inputArray	2	5	3	0	2	3	0	3

Diagram showing the calculation of the output array from the input array and count array. An arrow points from the value 3 at index 7 of the input array to a box containing 3. Another arrow points from this box to index 3 of the count array.

	0	1	2	3	4	5
countArray	2	2	4	7	7	8

Diagram showing the calculation of the output array. An arrow points from the value 7 at index 3 of the count array to a box containing  $7-1=6$ . Another arrow points from this box to index 6 of the output array.

	0	1	2	3	4	5	6	7
outputArray							3	

Counting Sort



### Step 12 :

	0	1	2	3	4	5	6	7
inputArray	2	5	3	0	2	3	0	3

Diagram showing the calculation of the count array. An arrow points from the value 2 at index 0 of the input array to a box containing 2. Another arrow points from this box to index 2 of the count array.

	0	1	2	3	4	5
countArray	0	2	3	4	7	7

Diagram showing the calculation of the output array. An arrow points from the value 2 at index 2 of the count array to a box containing 2. Another arrow points from this box to index 2 of the output array.

	0	1	2	3	4	5	6	7
outputArray	0	0	2	2	3	3	3	5

Counting Sort



# Bucket Sort

---

El algoritmo de Bucket Sort se aprovecha de la simplicidad de Insertion Sort y la utilidad de las Listas Enlazadas para ordenar elementos en forma contigua. Se asume que los valores de entrada están *uniformemente distribuidos* y tienen una *distribución aleatoria*. También nos gustaría que  $\forall x \in a \longrightarrow x \in [0, 1)$ .

La implementación de Bucket Sort consta de los siguientes pasos:

- Para cada elemento  $x$  del arreglo  $a$ , se multiplica por  $n$  y se coloca en la "cubeta" correspondiente, es decir, en la posición entera de ese resultado.
- Cada posición del arreglo actúa como una "cubeta" y contiene una lista enlazada que ordena los elementos que caen en esa cubeta.
- Para cada "cubeta", se utiliza el algoritmo Insertion Sort para ordenar los elementos dentro de la lista enlazada correspondiente.
- Habiendo ordenado las listas enlazadas, se itera ordenadamente a través de cada lista y se insertan los valores en una lista resultante.

*Aclaración personal:* en el caso de que no tengamos nuestros elementos en el dominio  $[0, 1)$ , podríamos realizar un paso extra que sería mapear esos valores al dominio. Tendríamos que verificar cuando devolvamos los elementos de "llevarlos" nuevamente a la base que pertenece.

**Input Array**

9.8	0.6	10.1	1.9	3.07	3.04	5.0	8.0	4.8	7.68
-----	-----	------	-----	------	------	-----	-----	-----	------



**Scatter in buckets**

**Unsorted Buckets**

0.6	1.9	3.07	3.04	5.0	4.8	8.0	7.68	9.8	10.1
-----	-----	------	------	-----	-----	-----	------	-----	------



**Sort Buckets**

**Sorted Buckets**

0.6	1.9	3.04	3.07	4.8	5.0	7.68	8.0	9.8	10.1
-----	-----	------	------	-----	-----	------	-----	-----	------



**Gather from buckets**

**Output Array**

0.6	1.9	3.04	3.07	4.8	5.0	7.68	8.0	9.8	10.1
-----	-----	------	------	-----	-----	------	-----	-----	------