



**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE COMPUTACIÓN
ESTRUCTURAS DE DATOS Y ALGORITMOS II**

TEMA I

ALGORITMOS DE ORDENAMIENTO

I ALGORITMOS DE ORDENAMIENTO

Objetivo: Comprender, analizar y comparar diversos algoritmos de ordenamiento.

I ALGORITMOS DE ORDENAMIENTO

I.I Ordenamiento.

I.I.I Bubble sort.

I.I.2 Merge sort.

I.I.3 Quick sort.

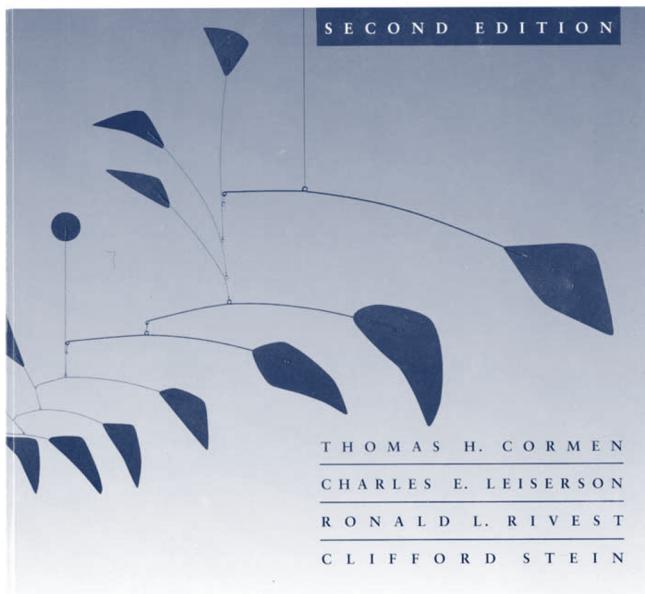
I.I.4 Heap sort.

I.I.5 Countig sort.

I.I.6 Radix sort.

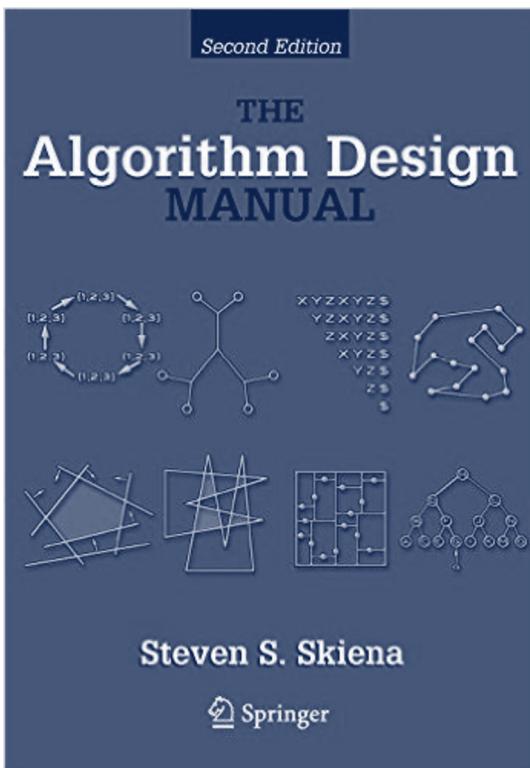
BIBLIOGRAFÍA

INTRODUCTION TO ALGORITHMS

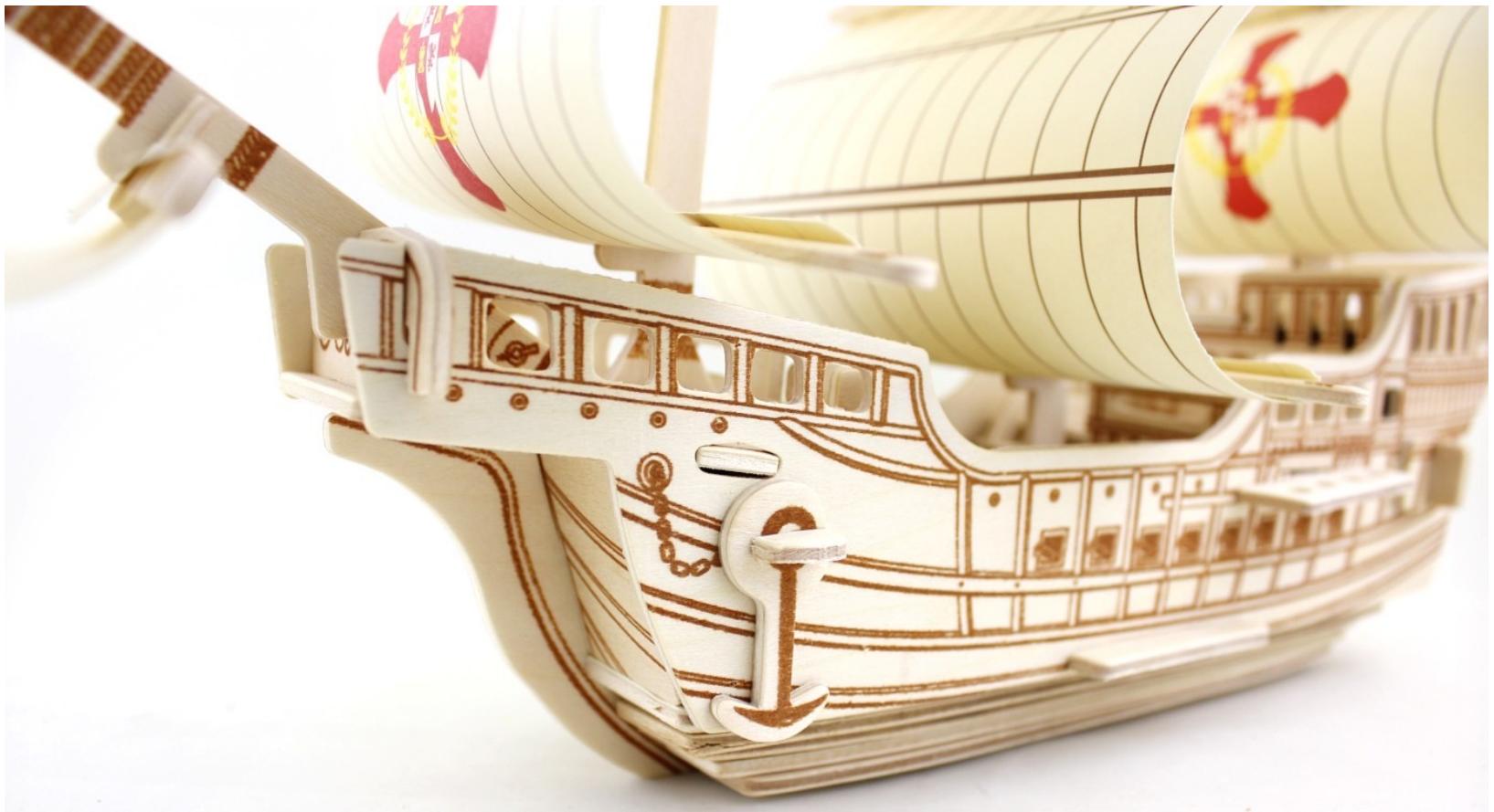


- **Introduction to Algorithms.** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, McGraw-Hill.

BIBLIOGRAFÍA



- The Algorithm Design Manual.
Steven S. Skiena, Springer.



I.I ORDENAMIENTO

12 horas

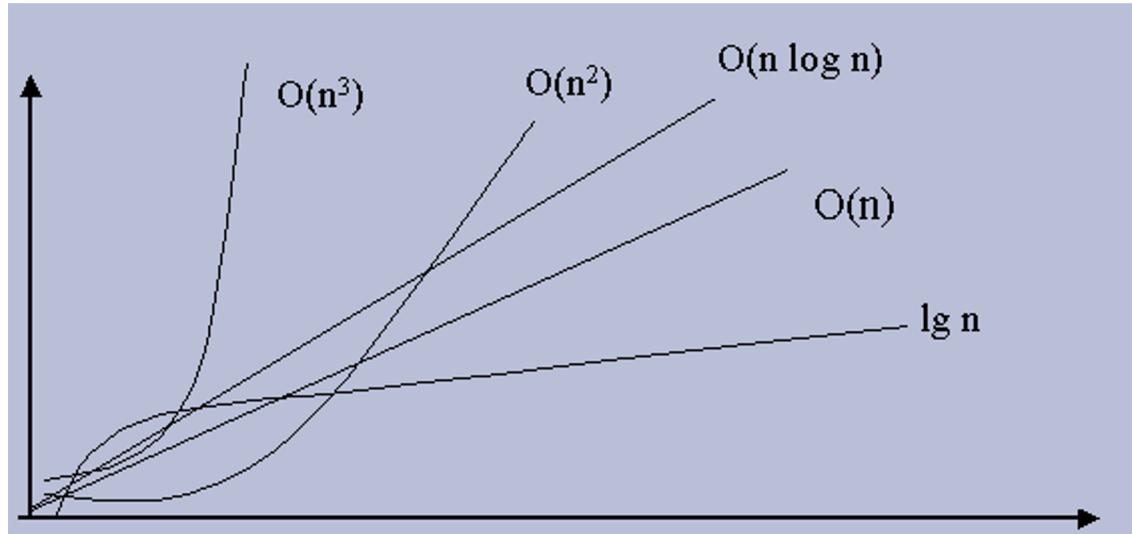


“The average user doesn’t give a damn what happens, as long as (1) it works and (2) it’s fast”

Daniel J. Bernstein
(German-American[2] mathematician, cryptologist, and programmer.)

¿ESTRUCTURAS DE DATOS LINEALES?





Notación O, omega y teta

3.3 Notación O, omega y teta

Las dos medidas más importantes para establecer la calidad de un algoritmo son:

- El tiempo total de ejecución, medido por el número de operaciones de cómputo realizadas durante el proceso.
- La cantidad de memoria utilizada por dicho proceso.



Tanto la eficiencia en la ejecución como la cantidad de memoria utilizada durante un proceso son parámetros que permiten medir la complejidad de un algoritmo.

Las técnicas más utilizadas para comparar la eficiencia de algoritmos sin necesidad de implementarlos son el modelo de computación RAM (Random Access Machine) y el análisis asintótico del peor caso de complejidad.



El modelo RAM se refiere a la representación de una computadora hipotética que permite evaluar la eficiencia del diseño de un algoritmo de manera independiente de la arquitectura (hardware) donde se implemente.

El modelo RAM asume las siguientes reglas:

- Cada operación simple (+, -, *, /, selección o llamada) toma exactamente un paso de tiempo.
- Los ciclos están compuestos por operaciones simples, por lo tanto, el tiempo que toma un ciclo depende del número de iteraciones del mismo.
- El acceso a memoria toma exactamente un paso de tiempo. La memoria de un modelo RAM es infinita.



El modelo RAM permite medir el tiempo que consume la ejecución de un algoritmo, contando el número de pasos que contiene el mismo. Si se asume que un modelo RAM ejecuta un cierto número de pasos por segundo, en automático se puede obtener el tiempo de ejecución de un proceso.

La RAM es un modelo simple que intenta representar el funcionamiento de un equipo, y, por tanto, hay que tener ciertas consideraciones.

En la mayoría de los procesadores, multiplicar dos números toma más tiempo que sumarlos. Esto viola la primera regla del modelo.

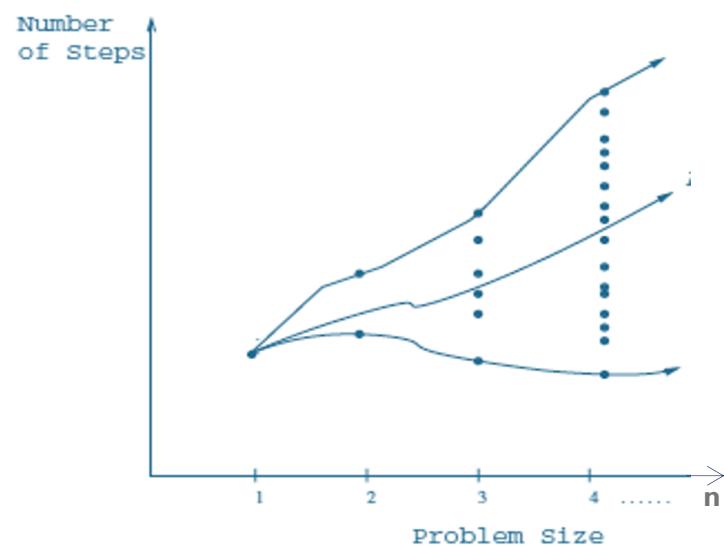
Además, los procesos paralelos o multihilos así como ciertos compiladores pueden violar la segunda regla.

Por si fuera poco, el acceso a la memoria difiere en tiempo dependiendo del lugar donde se encuentren los datos, con lo que se viola la tercera regla.



A pesar de los bemoles mencionados, el modelo RAM provee una excelente aproximación para entender cual sería el desempeño de un algoritmo en una computadora real. Es por lo anterior que es tan útil en la práctica.

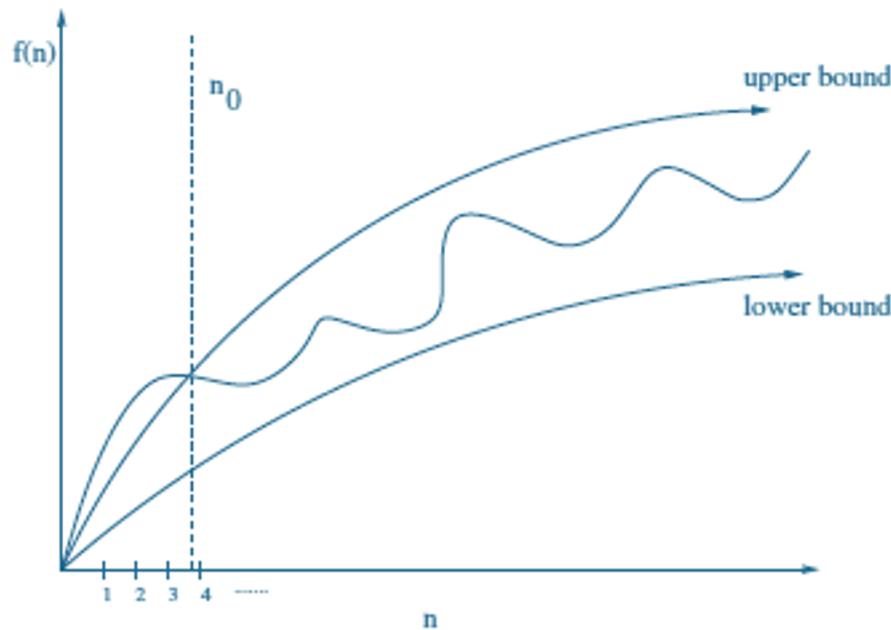
El tiempo de complejidad para un algoritmo dado es una función numérica que depende del tamaño de las instancias dadas (valores o datos de entrada).



Sin embargo, trabajar con las tres funciones o, más específicamente, con los casos extremos de las instancias para un algoritmo es bastante difícil debido a dos razones básicas:

- Las funciones poseen muchos saltos: instancias nones pueden requerir más tiempo que las pares, por ejemplo.
- Las funciones requieren mucho detalles para ser precisas:
$$t(n) = 12,754 n^2 + 4,353 n + 13,546$$

Por tanto, es mejor trabajar con funciones de tiempo de complejidad con límites superior e inferior. Esto se logra utilizando la notación O.





La notación O simplifica el análisis, ignorando niveles de detalle que no impacta en la comparación de algoritmos.

La notación O no hace diferencia entre multiplicación de constantes, es decir, la función $f(n) = 2 n$ y $g(n) = n$ son idénticas en el análisis O.

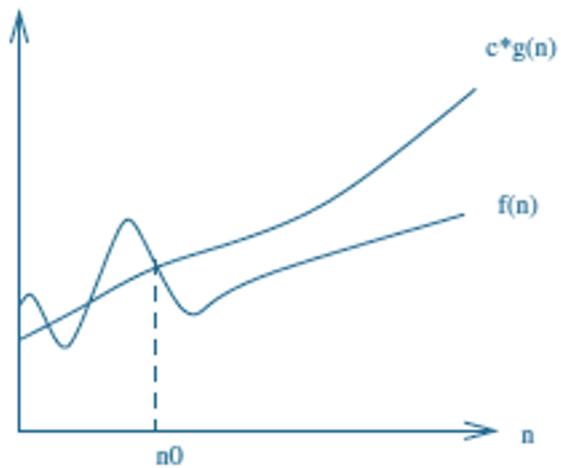
Las definiciones formales de la notación O son las siguientes:

- $f(n) = O(g(n))$ significa que $c*g(n)$ es el límite superior en $f(n)$. Por tanto, existe una constante c tal que $f(n)$ es siempre menor o igual a $c*g(n)$ por muy grande que n sea, para $n > n_0$.
- $f(n) = \Omega(g(n))$ significa que $c*g(n)$ es el límite inferior en $f(n)$. Por tanto, existe una constante c tal que $f(n)$ es siempre mayor o igual a $c*g(n)$ para $n > n_0$.

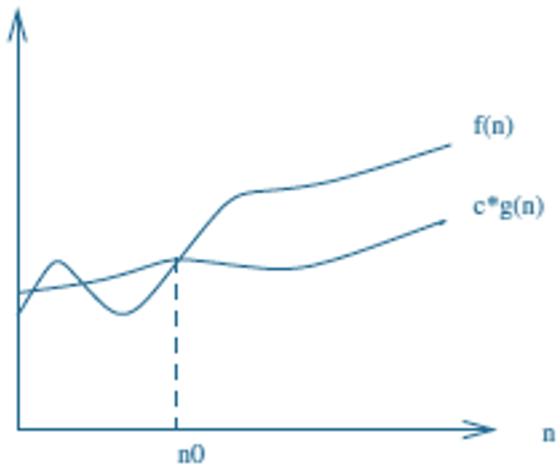
- $f(n) = \Theta(g(n))$ significa que $c_1*g(n)$ límite superior en $f(n)$ y $c_2*g(n)$ es un límite inferior en $f(n)$ para $n > n_0$. Por tanto, existen un par de constantes c_1 y c_2 tales que:

$$c_2*g(n) < f(n) < c_1*g(n)$$

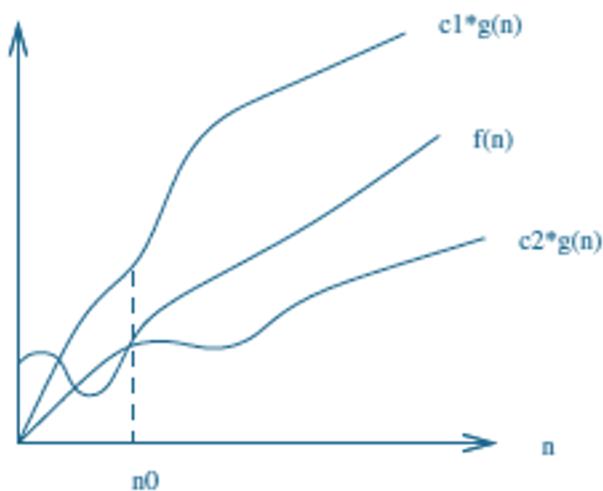
Lo que significa que $g(n)$ provee una muy estrecha y buena aproximación de $f(n)$.



$$f(n) = O(g(n))$$



$$f(n) = \Omega(g(n))$$



$$f(n) = \Theta(g(n))$$



La notación O (big oh) descarta los multiplicandos constantes dentro de una función.

Así, las funciones $f(n) = 0.001 n^2$ y $g(n) = 1000 n^2$ son tratadas de manera idéntica, a pesar de que $g(n)$ es un millón de veces mayor que $f(n)$ para cualquier valor de n .

A continuación se muestran las tasas de crecimiento en tiempo de las funciones más comunes, en una computadora donde cada operación toma un nano segundo 10^{-9} [s]:

n	$f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10		0.003 μ s	0.01 μ s	0.033 μ s	0.1 μ s	1 μ s	3.63 ms
20		0.004 μ s	0.02 μ s	0.086 μ s	0.4 μ s	1 ms	77.1 years
30		0.005 μ s	0.03 μ s	0.147 μ s	0.9 μ s	1 sec	8.4×10^{15} yrs
40		0.005 μ s	0.04 μ s	0.213 μ s	1.6 μ s	18.3 min	
50		0.006 μ s	0.05 μ s	0.282 μ s	2.5 μ s	13 days	
100		0.007 μ s	0.1 μ s	0.644 μ s	10 μ s	4×10^{13} yrs	
1,000		0.010 μ s	1.00 μ s	9.966 μ s	1 ms		
10,000		0.013 μ s	10 μ s	130 μ s	100 ms		
100,000		0.017 μ s	0.10 ms	1.67 ms	10 sec		
1,000,000		0.020 μ s	1 ms	19.93 ms	16.7 min		
10,000,000		0.023 μ s	0.01 sec	0.23 sec	1.16 days		
100,000,000		0.027 μ s	0.10 sec	2.66 sec	115.7 days		
1,000,000,000		0.030 μ s	1 sec	29.90 sec	31.7 years		

A partir de la tabla anterior se pueden obtener las siguientes conclusiones:

- Todos los algoritmos toman aproximadamente el mismo tiempo para $n=10$.
- El tiempo para cualquier algoritmo que ejecute $n!$ se vuelve inoperante para $n>20$.
- Para algoritmos con tiempo de ejecución 2^n tienen un rango operativo muy corto, ya que se vuelven imprácticos para $n>40$.

- Algoritmos cuyos tiempos de ejecución sean n^2 son rápidos mientras que $n < 10,000$, pero el tiempo se deteriora con rapidez con instancias más largas.
- Algoritmos lineales (n) o de la forma $(n * \log(n))$ son prácticos con instancias hasta de 1 billón.
- Un algoritmo de la forma $O(\log(n))$, mantienen un tiempo de ejecución bajo para cualquier valor de n .

Como se puede observar, a pesar de que el modelo RAM ignora los factores constantes de una función, el análisis brinda una muy buena aproximación de si un algoritmo es apropiado con base en el tamaño de los elementos de entrada.

Sin embargo, en la práctica un algoritmo cuyo tiempo de ejecución esté dado por $f(n) = n^3$ será más rápido que un algoritmo cuyo tiempo de ejecución sea $g(n) = 1,000,000 n^2$ cuando la instancia de entrada $n < 1,000,000$.



“Algorithmic complexity for structured programmers: All algorithms are $O(f(n))$, where f is someone else’s responsibility.”

Peter Cooper
(@peterc: Software development industry analyst.)

Dado el siguiente código, contabilizar el tiempo que se tarda el algoritmo siguiente en ejecutarse (las veces que se ejecuta cada sentencia):

```
FUNC doSomething() ENTONCES
    i ← 0 : ENTERO
    MIENTRAS i < 5 ENTONCES
        ESCRIBIR i
        i ← i + 1
    FIN MIENTRAS
    ESCRIBIR i
FIN FUNC
```

Dado el siguiente código, contabilizar el tiempo que se tarda el algoritmo siguiente en ejecutarse (las veces que se ejecuta cada sentencia):

	Tiempo
FUNC doSomething() ENTONCES	
i ← 0 : ENTERO	1
MIENTRAS i < 5 ENTONCES	6
ESCRIBIR i	5
i ← i + 1	5
FIN MIENTRAS	
ESCRIBIR i	1
FIN FUNC	

Para el ejemplo anterior, el número de veces que se ejecuta el ciclo es fijo (5 en este caso). Sin embargo, cuando el número de iteraciones está en función de una variable, el análisis del algoritmo utilizando los puntos que asume el modelo RAM permite generar el polinomio que describe su tiempo de ejecución.

Dado el siguiente algoritmo, obtener el polinomio que describe su comportamiento en función de la variable x.

```
FUNC countdown(x):VACIO
    y ← 0: ENTERO
    MIENTRAS x > 0 ENTONCES
        x ← x - 5
        y ← y + 1
    FIN MIENTRAS
    ESCRIBIR y
FIN FUNCIÓN
```

Dado el siguiente algoritmo, obtener el polinomio que describe su comportamiento en función de la variable x.

FUNC countdown(x):VACIO	Tiempo
y ← 0: ENTERO	
MIENTRAS x > 0 ENTONCES	$n/5 + $
x ← x - 5	$n/5$
y ← y + 1	$n/5$
FIN MIENTRAS	
ESCRIBIR y	
FIN FUNCIÓN	

El polinomio de la función es: $(3/5)*n + 3$

I.I ORDENAMIENTO

Los conjuntos son tan fundamentales para las ciencias de la computación como lo son para las matemáticas.

Los conjuntos manipulados por algoritmos pueden crecer, reducirse o cambiar en el tiempo, por lo tanto, se les denomina conjuntos dinámicos.



Dentro de los conjuntos dinámicos se pueden realizar diversas operaciones que se pueden agrupar en dos categorías: consultas y modificaciones.

Las consultas regresan información del conjunto. Las modificaciones pueden cambiar los elementos del conjunto.

La siguiente es una lista de operaciones típicas dentro de un conjunto de elementos:

- insert (S, x).
- delete (S, x).
- search (S, k).
- minimum (S).
- maximum (S).
- successor (S, x).
- predecessor (S, x).

El tiempo que toma ejecutar una operación en un conjunto S es proporcional al tamaño de entrada del conjunto (instancias de entrada).



Dentro de las estructuras de datos, el término ordenar significa reagrupar o reorganizar un conjunto de datos u objetos en una secuencia específica.

Por tanto, la clasificación (ordenación) de la información permite realizar búsquedas y/o actualización de información dentro de un conjunto de datos de una forma más eficiente.

De manera formal, un ordenamiento se define a partir de un conjunto de elementos. Sea A un conjunto de N elementos:

$$A_1, A_2, A_3, \dots, A_n$$

Ordenar significa permutar los elementos de tal forma que se clasifiquen de acuerdo a una distribución preestablecida.

- Ascendente $A_1 \leq A_2 \leq A_3 \leq \dots \leq A_n$
- Descendente $A_1 \geq A_2 \geq A_3 \geq \dots \geq A_n$



Métodos de ordenamiento

Los métodos de ordenamiento se dividen en dos grupos: ordenamientos internos y ordenamientos externos.

- Ordenamientos internos: son aquellos que se realizan en la memoria principal.
- Ordenamientos externos: son aquellos que se realizan tanto en memoria principal como en memoria secundaria.

Tipos de ordenamiento

Ordenamiento interno

Por selección

Por intercambio

Por inserción

Por distribución

Por intercalación

Ordenamiento externo

Por polifase

Por cascada

Oscilantes

Por distribución



Los métodos de ordenamiento se pueden clasificar con base en las acciones internas del algoritmo en los siguientes tipos:

- Método de inserción
- Método de intercalación
- Método de intercambio
- Método de distribución
- Método de selección

Método de inserción

El método consiste en insertar un elemento del conjunto en la parte izquierda (o derecha, según se decida) del mismo, que ya se encuentra ordenado.

Los ordenamientos que ocupan este tipo de algoritmos son inserción directa, shell sort, inserción binaria y función hash.

Método de intercalación

Este tipo de algoritmos consisten en unir dos o más grupos de elementos en un solo conjunto. Los conjuntos a unir deben estar previamente ordenados para poder realizar la intercalación. Por tanto, no se pueden intercalar conjuntos ordenados de manera inversa (uno ascendente y otro descendente o viceversa) y, mucho menos, conjuntos desordenados.

El algoritmo más representativo de este tipo de métodos es merge sort.

Método de intercambio

El algoritmo consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados, partiendo del extremo izquierdo o del extremo derecho, según se deseé.

Los ordenamientos más representativos que ocupan este tipo de algoritmos son bubble sort y quick sort.

Método de distribución

Consiste en una colección de casilleros donde se deposita el elemento que coincide con la marca del casillero.

El algoritmo más representativo de este tipo de métodos es radix sort.

Método de selección

El algoritmo consiste en buscar dentro del conjunto el elemento menor (o mayor) para colocarlo en la primera posición (o última posición), repitiendo el proceso hasta ordenar todo el conjunto.

El algoritmo más representativo de este tipo de métodos es selection sort.

Clasificación por complejidad

Así mismo, los métodos de ordenamiento se pueden clasificar con base en su complejidad en:

- Métodos directos (n^2): tienen como característica que su implementación es relativamente sencilla, pero son ineficientes cuando la instancia de entrada es grande ($n>10,000$).
- Métodos logarítmicos ($n*\log(n)$): son algoritmos difíciles de implementar, pero con mejor eficiencia.



La eficiencia de un algoritmo está definida por el tiempo de ejecución del mismo. Dentro de los métodos de ordenamiento, el tiempo de ejecución depende, fundamentalmente, del número de comparaciones y del número de movimientos que se realicen para clasificar los elementos.

Por tanto, cuando la instancia de entrada es pequeña, se recomienda utilizar métodos directos, cuando la instancia de entrada es grande es más recomendable utilizar métodos logarítmicos.



BUBBLE SORT

I.I.I BUBBLE SORT

El ordenamiento de la burbuja (bubble sort) es un método de intercambio directo. Es uno de los métodos de ordenamiento más simples tanto en su comprensión como en su implementación, aunque es un método muy ineficiente.

Este método puede trabajar de dos maneras distintas: moviendo los elementos más pequeños a la izquierda del conjunto o trasladando los elementos más grandes a la derecha del conjunto (para realizar un acomodo ascendente).



El algoritmo consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados, partiendo del extremo izquierdo o derecho, según se deseé.

Por tanto, se realizan $n-1$ comparaciones para mover el elemento menor o mayor (según sea el caso) a su posición final.

Ordenar el siguiente conjunto, transportando el elemento de menor tamaño hacia la parte izquierda del conjunto, utilizando bubble sort.

A



Se realiza el primer recorrido.

A



$A[7]>A[8]$	$12 > 35$	No hay intercambio.
$A[6]>A[7]$	$27 > 12$	Sí hay intercambio.
$A[5]>A[6]$	$44 > 12$	Sí hay intercambio.
$A[4]>A[5]$	$16 > 12$	Sí hay intercambio.
$A[3]>A[4]$	$8 > 12$	No hay intercambio.
$A[2]>A[3]$	$67 > 8$	Sí hay intercambio.
$A[1]>A[2]$	$15 > 8$	Sí hay intercambio.

A



Se realiza el segundo recorrido.

A



$A[7]>A[8]$	$27 > 35$	No hay intercambio.
$A[6]>A[7]$	$44 > 27$	Sí hay intercambio.
$A[5]>A[6]$	$16 > 27$	No hay intercambio.
$A[4]>A[5]$	$12 > 16$	No hay intercambio.
$A[3]>A[4]$	$67 > 12$	Sí hay intercambio.
$A[2]>A[3]$	$15 > 12$	Sí hay intercambio.

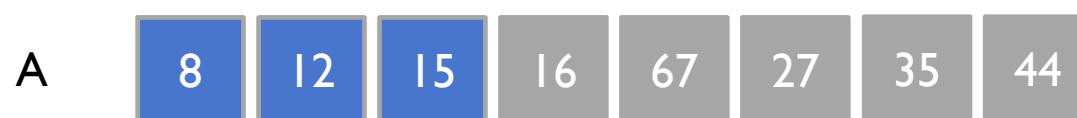
A



Se realiza el tercer recorrido.



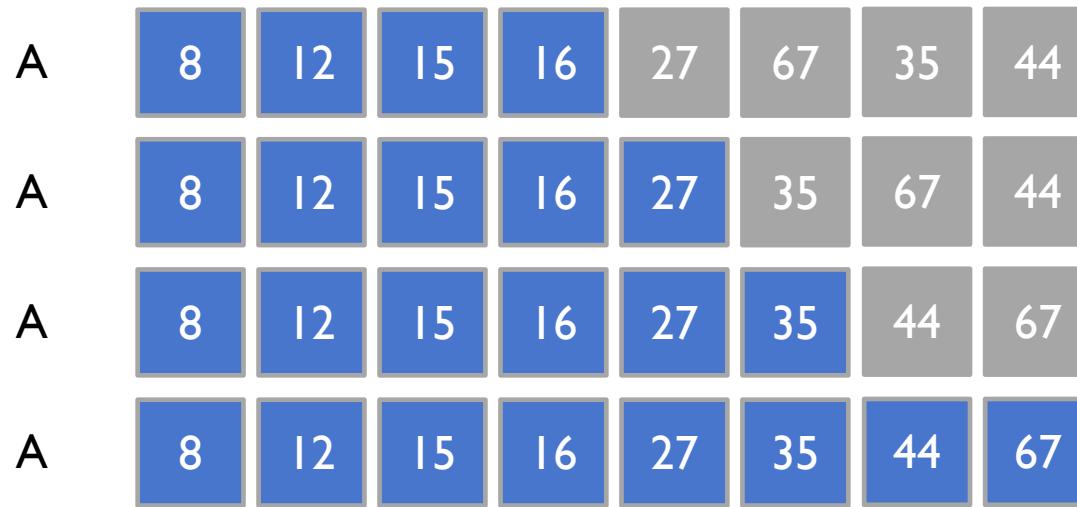
$A[7]>A[8]$	$44 > 35$	Sí hay intercambio.
$A[6]>A[7]$	$27 > 35$	No hay intercambio.
$A[5]>A[6]$	$16 > 27$	No hay intercambio.
$A[4]>A[5]$	$67 > 16$	Sí hay intercambio.
$A[3]>A[4]$	$15 > 16$	No hay intercambio.



El recorrido y las comparaciones se realizan $n-1$ veces.

A	15	67	8	16	44	27	12	35
A	8	15	67	12	16	44	27	35
A	8	12	15	67	16	27	44	35
A	8	12	15	16	67	27	35	44
A	8	12	15	16	27	67	35	44

El recorrido y las comparaciones se realizan $n-1$ veces.





BUBBLE SORT



“Weeks of coding can save you hours of planning.” - Unknown

@CodeWisdom

El algoritmo de ordenamiento utilizando el método de la burbuja es el siguiente:

```
FUNC BUBBLE_SORT(A[ ]: Entero): DEV vacío
    cont ← 0, comp ← 0, aux ← 0: ENTERO
    MIENTRAS cont < length(A)-1 HACER
        comp ← length(A)-1
        MIENTRAS comp > cont HACER
            SI A[comp-1] > A[comp] HACER
                aux ← A[comp-1]
                A[comp-1] ← A[comp]
                A[comp] ← aux
            FIN_SI
            comp -= 1
        FIN_MIENTRAS
        cont += 1
    FIN_MIENTRAS
FIN_FUNC
```

El análisis de complejidad del algoritmo de ordenación utilizando el método de la burbuja es el siguiente:

FUNC BUBBLE_SORT(A[]): Entero): DEV vacío	costo	iteraciones
cont \leftarrow 0, comp \leftarrow 0, aux \leftarrow 0: ENTERO	c_1	1
MIENTRAS cont < length(A)-1 HACER	c_2	n
comp \leftarrow length(A)-1	c_3	n-1
MIENTRAS comp > cont HACER	c_4	$\sum_{comp=0}^n t_{comp}$
SI A[comp-1] > A[comp] HACER	c_5	$\sum_{comp=0}^n t_{comp} - 1$
aux \leftarrow A[comp-1]	c_6	$\sum_{comp=0}^n t_{comp} - 1$
A[comp-1] \leftarrow A[comp]	c_7	$\sum_{comp=0}^n t_{comp} - 1$
A[comp] \leftarrow aux	c_8	$\sum_{comp=0}^n t_{comp} - 1$
FIN_SI		
cmp -= 1	c_9	$\sum_{comp=0}^n t_{comp} - 1$
FIN_MIENTRAS		
cont += 1	c_{10}	n-1
FIN_MIENTRAS		
FIN_FUNC		



Si el conjunto se encuentra en forma inversa a como se desea ordenar, entonces se presenta el peor caso de complejidad y, por tanto, se deben comparar todos los elementos $A[j]$ (ciclo externo) con cada elemento del subarreglo (ciclo interno).

Por inducción matemática, la sumatoria de $[2, n]$ de t_j y t_{j-1} son, respectivamente:

$$\sum_{j=2}^n t_j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^n (t_j - 1) = \frac{n(n-1)}{2}$$

Por lo tanto, para el peor caso, el tiempo de ejecución del algoritmo de ordenamiento es:

$$T(n) = c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 (((n*(n+1))/2)-1) + \\ + c_5 ((n*(n-1))/2) + c_6 ((n*(n-1))/2) + c_7(n - 1)$$

$$T(n) = (c_4/2 + c_5/2 + c_6/2)*n^2 \\ + (c_1+c_2+c_3+c_4/2+c_5/2+c_6/2+c_7)*n \\ + (c_2+c_3+c_4+c_7)$$

De aquí se obtiene que el tiempo de ejecución del peor caso de complejidad se puede expresar con el polinomio: $an^2 + bn + c$.

Si el conjunto se encuentra dispuesto en la misma forma en la que se desea ordenar, se tiene el mejor caso de complejidad. ¿Qué complejidad tiene?

```
FUNC BUBBLE_SORT(A[ ]: Entero): DEV vacío
    cont ← 0, comp ← 0, aux ← 0: ENTERO
    MIENTRAS cont < length(A)-1 HACER
        comp ← length(A)-1
        MIENTRAS comp > cont HACER
            SI A[comp-1] > A[comp] HACER
                aux ← A[comp-1]
                A[comp-1] ← A[comp]
                A[comp] ← aux
            FIN_SI
            cmp -= 1
        FIN_MIENTRAS
        cont += 1
    FIN_MIENTRAS
FIN_FUNC
```

Un conjunto aleatorio representa el caso promedio de complejidad. El análisis de este caso es importante, porque representa la constante en la práctica, es decir, la mayoría de los conjuntos a ordenar son aleatorios.

```
FUNC BUBBLE_SORT(A[ ]: Entero): DEV vacío
    cont ← 0, comp ← 0, aux ← 0: ENTERO
    MIENTRAS cont < length(A)-1 HACER
        comp ← length(A)-1
        MIENTRAS comp > cont HACER
            SI A[comp-1] > A[comp] HACER
                aux ← A[comp-1]
                A[comp-1] ← A[comp]
                A[comp] ← aux
            FIN_SI
            cmp -= 1
        FIN_MIENTRAS
        cont += 1
    FIN_MIENTRAS
FIN_FUNC
```



Una vez realizado el análisis teórico es necesario realizar la comprobación empírica de la función obtenida.

Para ello se deben graficar los tiempos de ejecución del algoritmo para diferentes instancias de entrada.

El algoritmo de ordenamiento BubbleSort tiene como instancia de entrada una lista de tamaño n y se requiere que el algoritmo regrese el tiempo (número de pasos) que se tarda en ejecutar (ordenar) la lista dada.

Python posee la función time(), la cual funciona como un cronómetro que acciona el propio intérprete y que, por lo mismo, no es tan confiable.

Para las prácticas se va a contabilizar el número de veces que un algoritmo se ejecuta utilizando un contador en lugar de la función time() de Python.

A partir del algoritmo de ordenamiento probado (el cual ordena correctamente), ¿qué se debe agregar para contabilizar el tiempo de ejecución?

```
FUNC BUBBLE_SORT(A[ ]: Entero): DEV vacío
    cont ← 0, comp ← 0, aux ← 0: ENTERO
    MIENTRAS cont < length(A)-1 HACER
        comp ← length(A)-1
        MIENTRAS comp > cont HACER
            SI A[comp-1] > A[comp] HACER
                aux ← A[comp-1]
                A[comp-1] ← A[comp]
                A[comp] ← aux
            FIN_SI
            cmp -= 1
        FIN_MIENTRAS
        cont += 1
    FIN_MIENTRAS
FIN_FUNC
```

A partir del algoritmo de ordenamiento probado (el cual ordena correctamente), ¿qué se debe agregar para contabilizar el tiempo de ejecución?

```
FUNC BUBBLE_SORT(A[ ]: Entero): DEV vacío
    times ← 0: ENTERO
    cont←0, comp ← 0, aux ← 0: ENTERO
    MIENTRAS cont < length(A)-1 HACER
        times += 1
        comp ← length(A)-1
        MIENTRAS comp > cont HACER
            times += 1
            SI A[comp-1] > A[comp] HACER
                aux ← A[comp-1]
                A[comp-1]←A[comp]
                A[comp]←aux
            FIN_SI
            cmp -= 1
        FIN_MIENTRAS
        cont += 1
    FIN_MIENTRAS
    DEV times
FIN_FUNC
```

El algoritmo anterior contabiliza el tiempo que se tarda en ordenar una lista de tamaño n . Para realizar la gráfica se requieren generar m listas de tamaño variable (desde 1 hasta n) y, para cada lista generada, se debe guardar el tiempo que se tarda en realizar el ordenamiento.

Las listas pueden estar en:

- El orden que se desea ordenar (mejor caso de complejidad)
- En orden inverso al que se desea ordenar (peor caso de complejidad)
- En orden aleatorio (caso promedio de complejidad).

BUBBLE SORT CON SEÑAL

Este método es una modificación del método de la burbuja. Consiste en utilizar una bandera (señal) que permita validar si se ha o no efectuado algún intercambio, esto con el fin de comprobar si el conjunto ya se encuentra ordenado y no seguir recorriendo el mismo.

El algoritmo de ordenamiento utilizando el método de la burbuja con señal es el siguiente:

```
FUNC BUBBLE_SORT_SIGNAL(A[]): Entero; DEV vacío
    cont ← 0, comp ← 0, aux ← 0: ENTERO, signal ← false: BOOL
    MIENTRAS cont < length(A)-1 Y signal = false HACER
        comp ← length(A)-1, signal ← true
        MIENTRAS comp > cont HACER
            SI A[comp-1] > A[comp] HACER
                aux ← A[comp-1]
                A[comp-1] ← A[comp]
                A[comp] ← aux
                signal ← false
            FIN_SI
            cmp -= 1
        FIN_MIENTRAS
        cont += 1
    FIN_MIENTRAS
FIN_FUNC
```



“If it doesn’t work, it doesn’t matter how fast it doesn’t work.”

Mich Ravera
(Software Developer and Publisher, Bridge Director & Player, Private Pilot)



MERGE SORT

I.I.6 MERGE SORT

El algoritmo de unión ordenada (merge sort) permite combinar dos conjuntos de elementos previamente ordenados, siguiendo el paradigma divide y vencerás:

- Divide la secuencia de n elementos a ser ordenados en dos subsecuencias de $n/2$ elementos cada una.
- Ordena recursivamente las dos subsecuencias de elementos utilizando unión ordenada.
- Combina (une) las dos subsecuencias ordenadas para producir la salida ordenada.

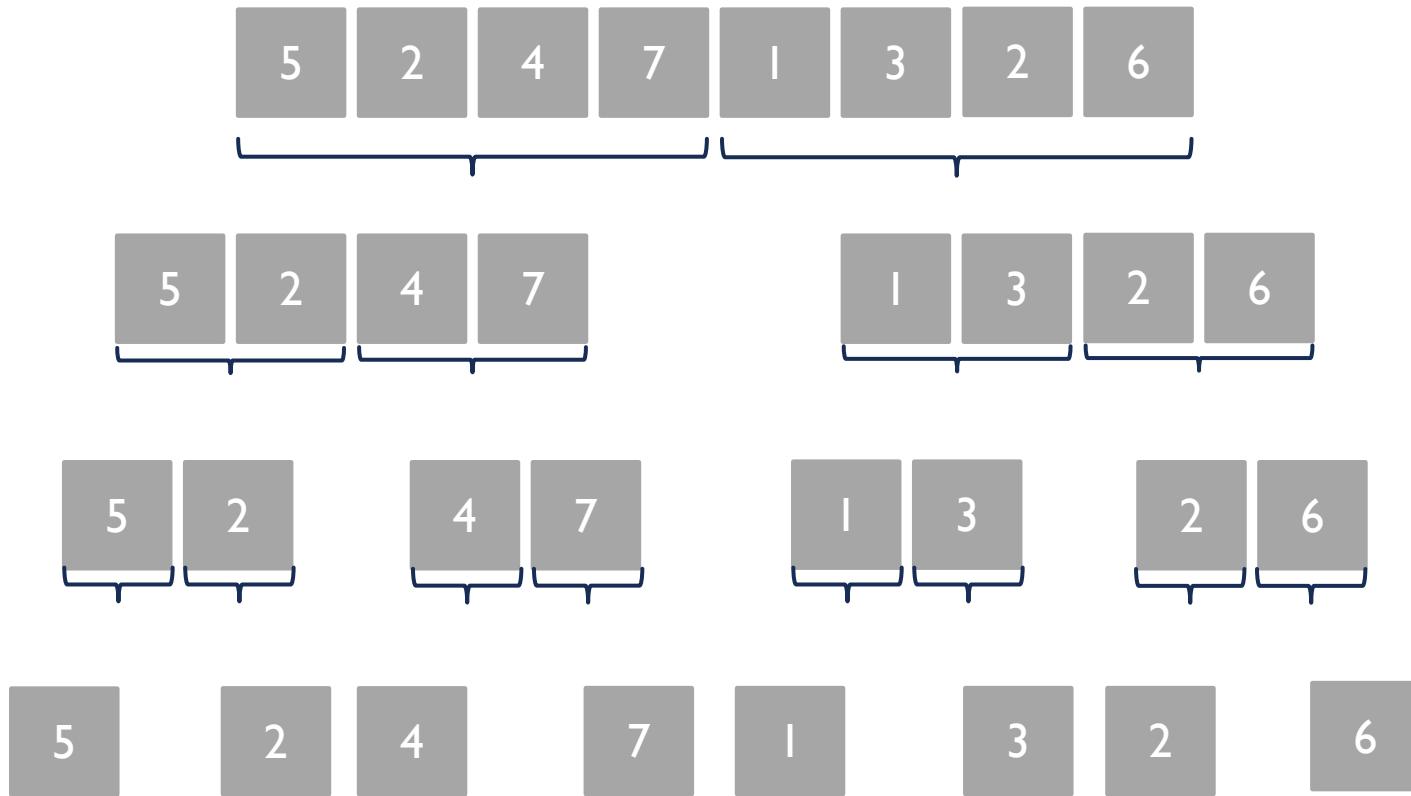
El algoritmo de unión ordenada `merge_sort(A, p, r)` recibe como parámetros un arreglo de elementos, un índice inferior y un índice superior, y acomoda los elementos en un subarreglo $A[p, r]$.

Si $p > r$, el arreglo tendrá como máximo un elemento y, por tanto, ya se encuentra ordenado. De lo contrario, se divide el arreglo $A[p, r]$ en $A[p, q]$ y $A[q+1, r]$ que contienen, respectivamente, la mitad de los elementos del arreglo ($n/2$).

Ordenar el siguiente conjunto, transportando el elemento de menor tamaño hacia la parte izquierda del conjunto, utilizando merge sort.

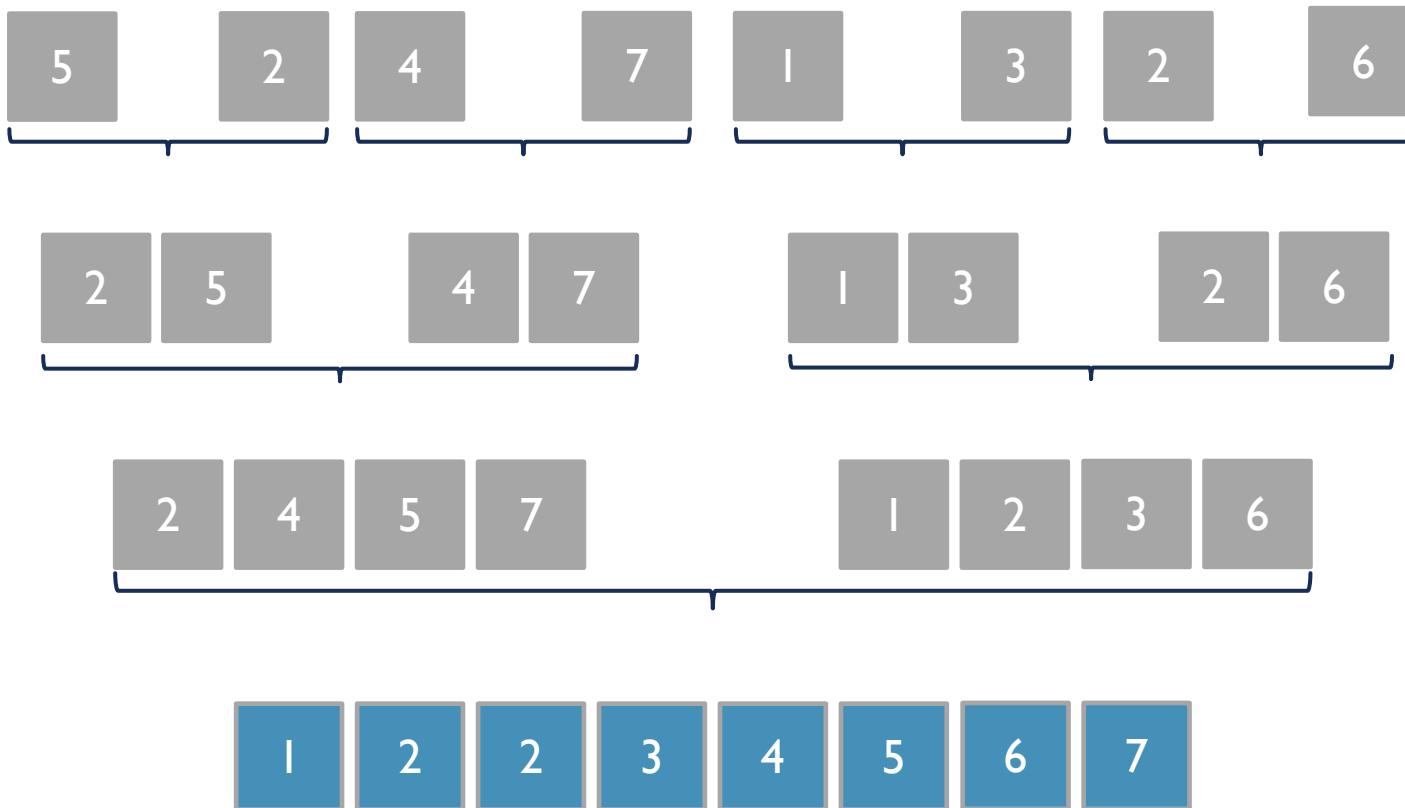
A





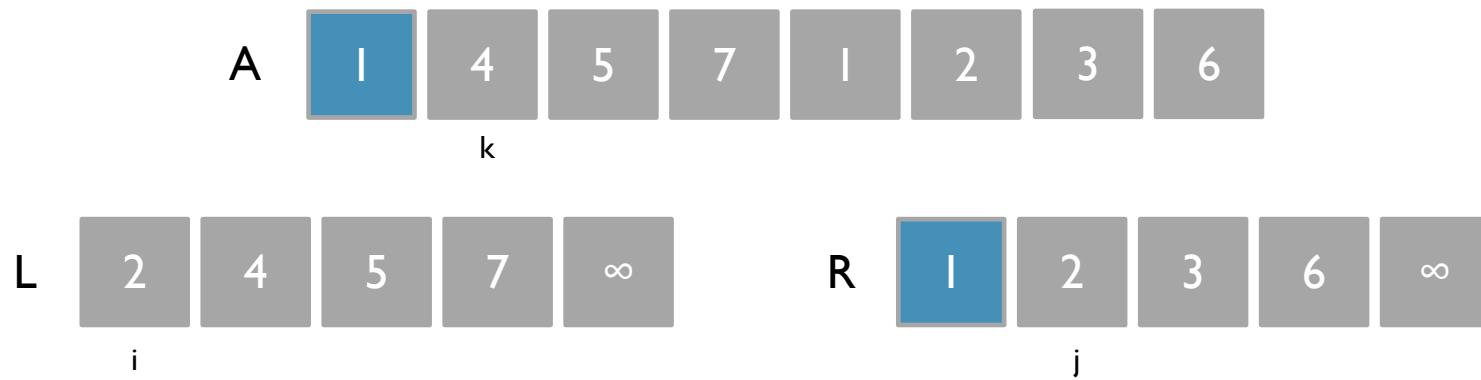
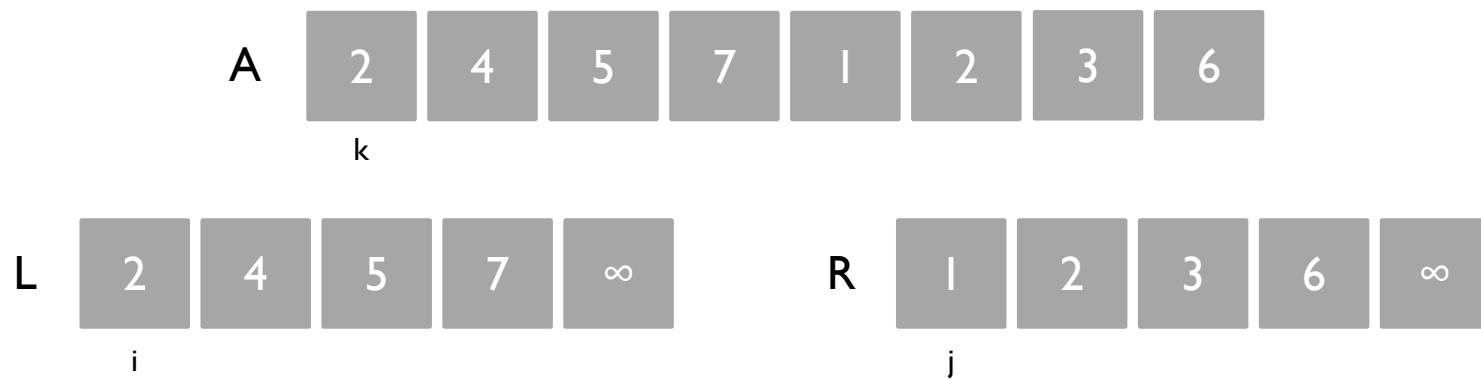
La última operación del algoritmo consiste en unir las dos secuencias ordenadas en un paso combinado.

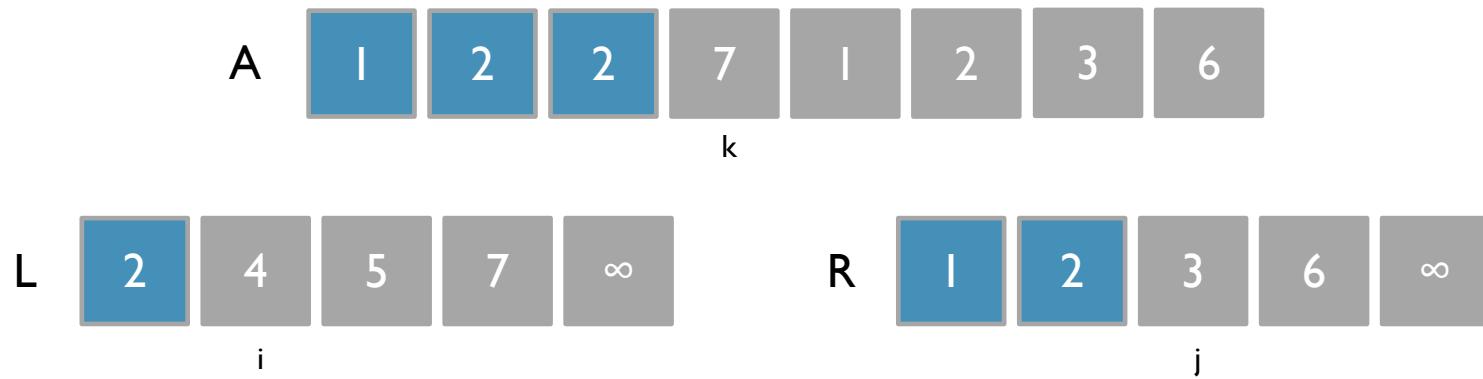
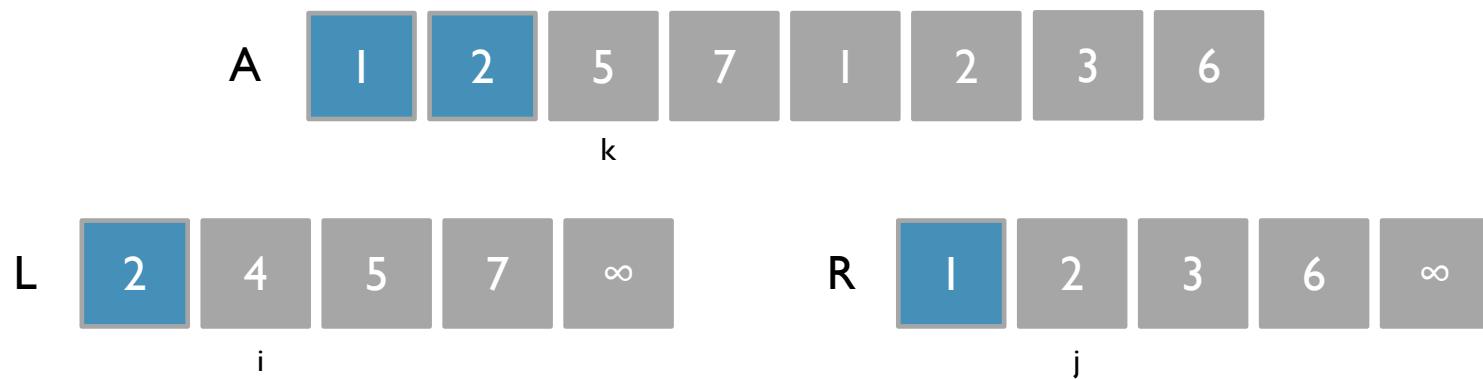
Para realizar dicha unión se utiliza una función auxiliar $\text{unir}(A, p, q, r)$, donde A es un arreglo, p , q y r son índices del arreglo tales que $p < q < r$. La función asume que los subarreglos $A[p, q]$ y $A[q+1, r]$ están previamente ordenados. El proceso une los dos arreglos para formar un subarreglo ordenado que reemplaza a $A[p, r]$.

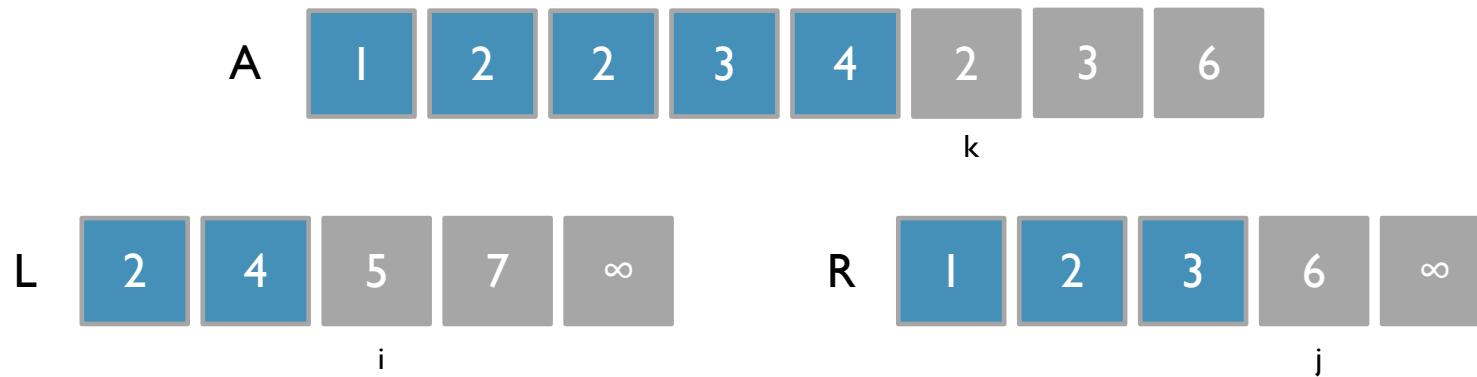
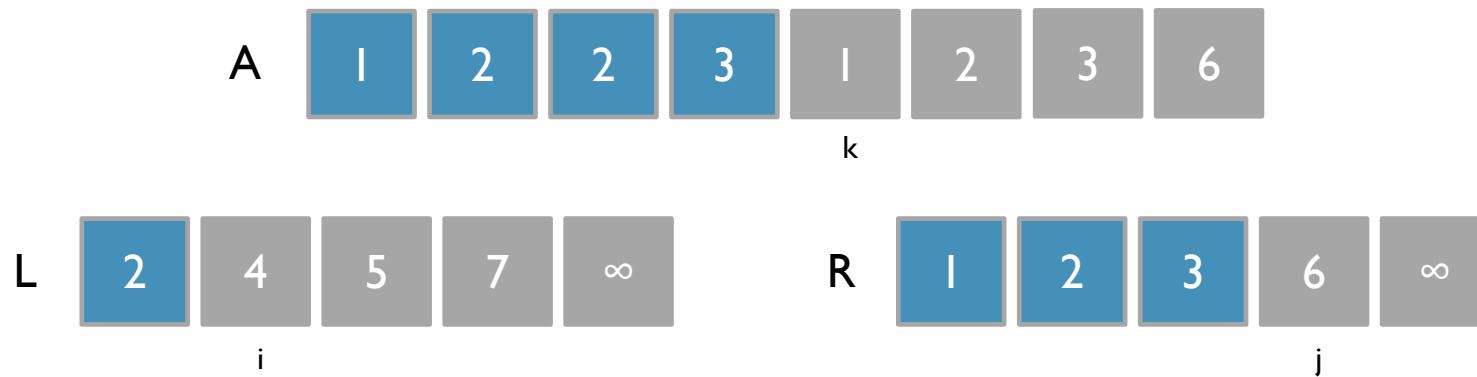


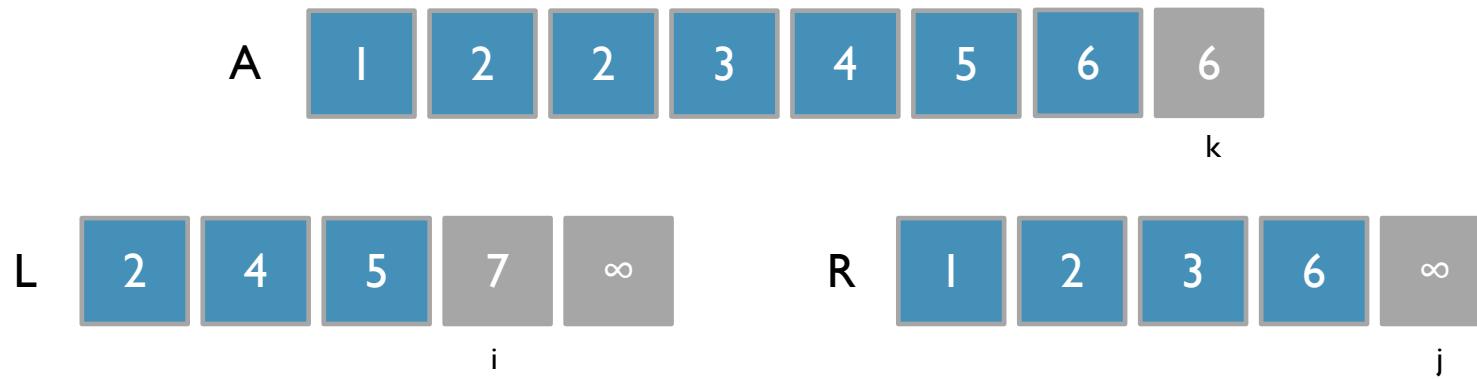
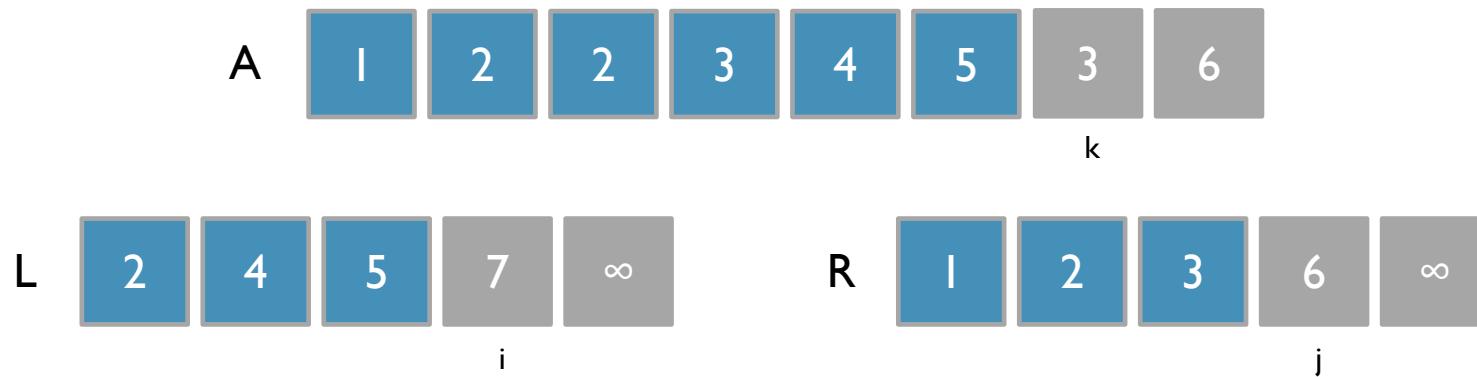
El algoritmo de unión consiste en unir los subconjuntos $A[p, q]$ y $A[q+1, r]$ que están previamente ordenados.

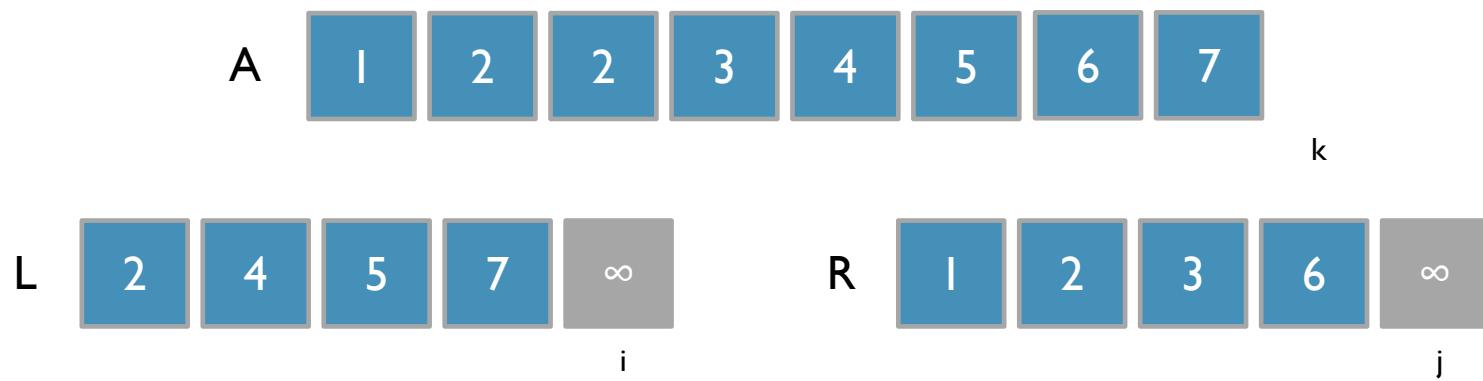














MERGE SORT

b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8] b[9]

El algoritmo que permite realizar la unión ordenada de los elementos es el siguiente:

```
FUNC MERGE-SORT (A, p, r)
    SI p < r ENTONCES
        q ← (p + r)/2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
    FIN_SI
FIN_FUNC
```

El algoritmo de auxiliar que une el conjunto previamente ordenado es el siguiente:

```
FUNC MERGE (A[]:ENT, p:ENT, q:ENT, r:ENT)
    n1 ← q - p
    n2 ← r - q
    create_arrays L[1...n1 + 1] and R[1...n2 + 1]
    MIENTRAS i ← 1 <= n1
        L [ i ] ← A[p + i - 1]
    FIN_MIENTRAS
    MIENTRAS j ← 1 <= n2
        R [ j ] ← A[q + j ]
    FIN_MIENTRAS
    L [n1 + 1] ← ∞
    R [n2 + 1] ← ∞
    i ← 1
    j ← 1
    MIENTRAS k ← p <= r
        SI j >= r-q O (i < q-p+1 Y L[i ] ≤ R[ j ])
            A[k] ← L[i ]
            i ← i + 1
        FIN_SI
        DE_LO CONTRARIO
            A[k] ← R[ j ]
            j ← j + 1
        FIN_DE_LO CONTRARIO
        k ← k + 1
    FIN_MIENTRAS
FIN_FUNC
```



ALGORITMOS DE ORDENAMIENTO. PARTE I.

PRÁCTICA I

ALGORITMOS DE ORDENAMIENTO. PARTE I.

- Implementar los algoritmos de Bubble sort y Merge sort en Python.
- Obtener los polinomios del mejor, el peor y el caso promedio de complejidad de cada algoritmo (Bubble sort y Merge sort).
- Graficar el comportamiento de los algoritmos para diferentes instancias de tiempo (listas de 1 a 1000 elementos) para el mejor (lista ordenada), el peor (lista ordenada en forma inversa) y el caso promedio (lista aleatoria) de complejidad.



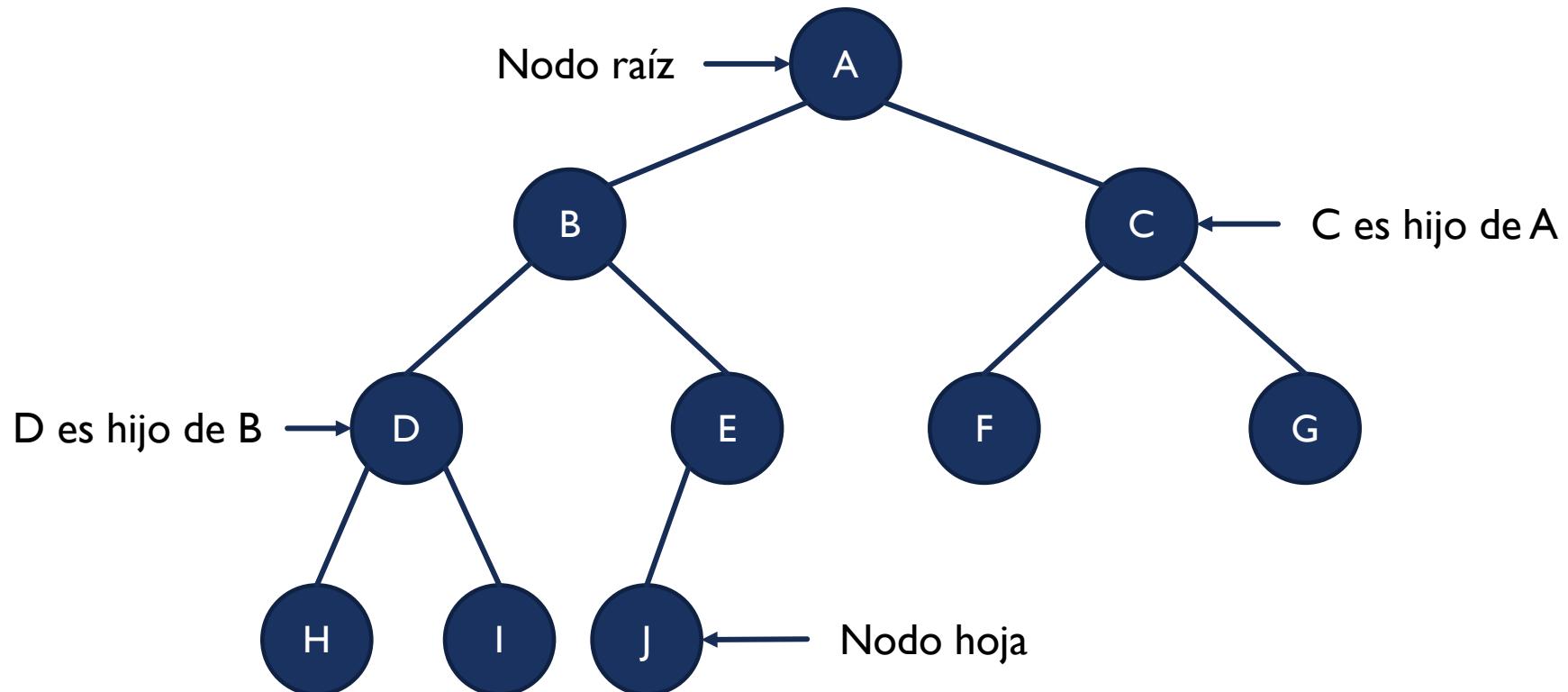
HEAP SORT

I.I.2 HEAP SORT

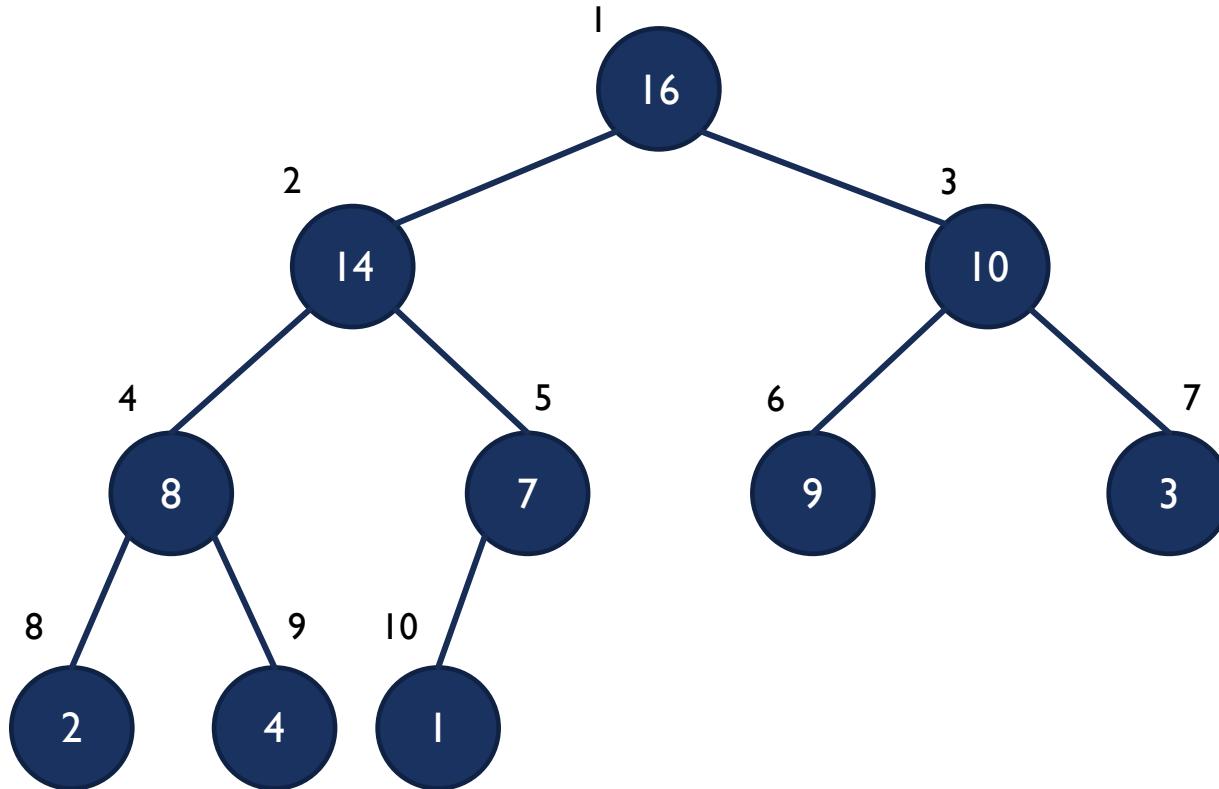
La estructura de datos montículo binario (binary heap) es un arreglo de objetos, el cual puede ser visto como un árbol binario. Cada nodo del árbol corresponde a un elemento en el arreglo donde se almacena su valor.

El montículo se construye desde el nodo inicial (raíz) hasta los nodos finales (hoja). Es posible que un montículo no tenga todos los nodos finales.

Un árbol binario tiene las siguientes características:



Un montículo binario está completamente lleno en todos sus niveles excepto el más bajo, el cual puede o no estar lleno.



Un montículo binario (al igual que un árbol binario) se puede representar a través de un arreglo, donde el iésimo elemento (nodo) tiene a su hijo izquierdo en la posición $2 * i$ y a su hijo derecho en la posición $2 * (i + 1)$.





Por su disposición, los montículos se pueden clasificar en montículo máximo y en montículo mínimo.

Un montículo máximo cumple con la propiedad de que para cualquier nodo i : $A[\text{PARENT}(i)] \geq A[i]$. Es decir, el nodo i ésimo tiene como valor máximo el valor del padre, por tanto, el valor máximo del montículo se encuentra en la raíz.



Un montículo mínimo cumple con la propiedad de que para cualquier nodo i : $A[\text{PARENT}(i)] \leq A[i]$. Es decir, el nodo i ésimo tiene como valor mínimo el valor del padre, por tanto, el valor mínimo del montículo se encuentra en la raíz.

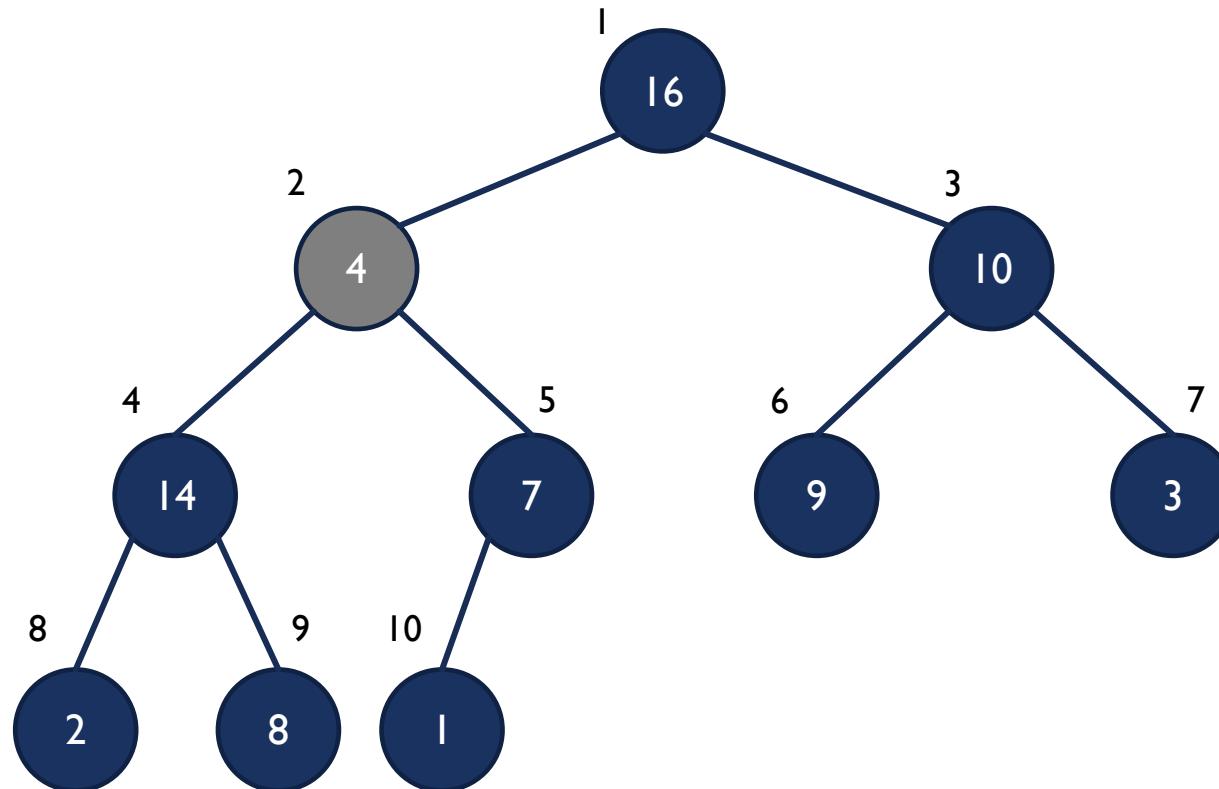
El algoritmo heap sort se basa en un montículo máximo.

El algoritmo heap sort parte de la afirmación de que se posee un montículo máximo para poder ordenar los elementos.

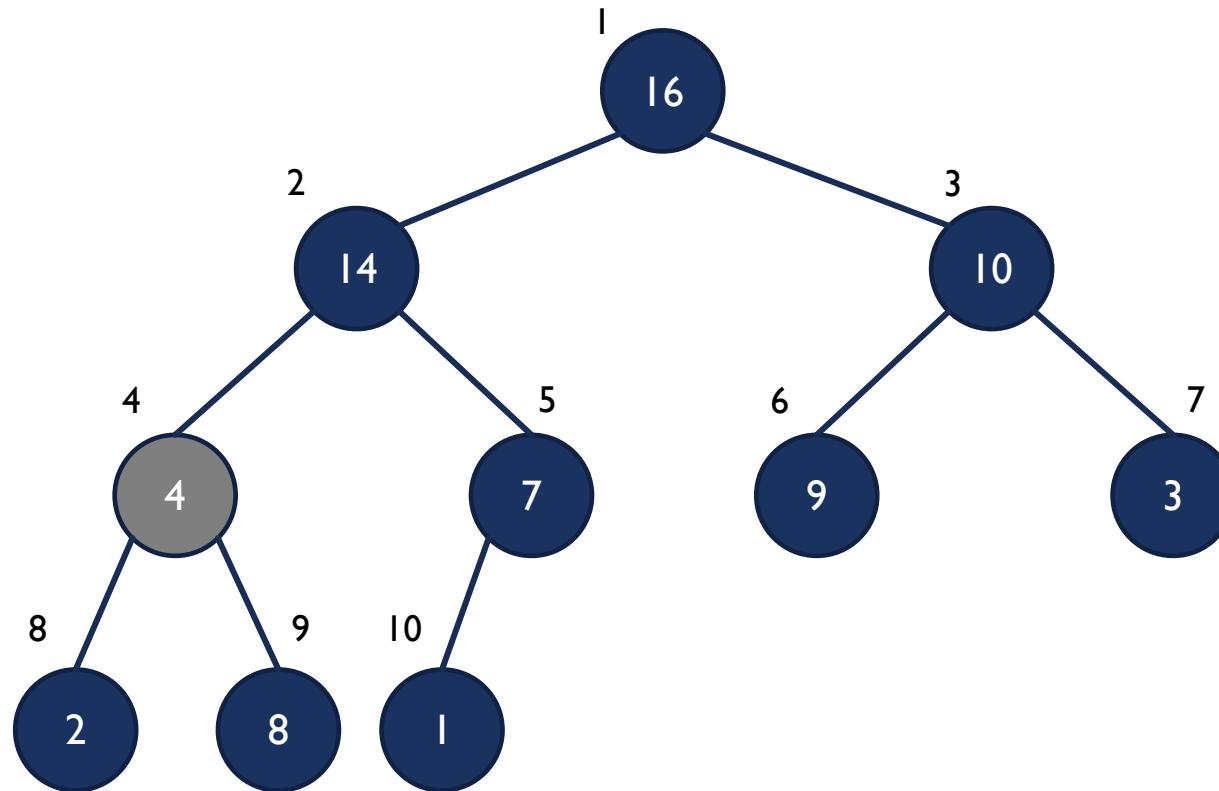
La función MAX-HEAPIFY permite conservar la propiedad del máximo de un montículo. Recibe como entradas el conjunto de datos A y el índice i donde se encuentra el nodo a acomodar.

MAX-HEAPIFY asume que LEFT(i) y RIGHT(i) son montículos máximos, pero considera que A[i] puede ser menor que sus hijos. El algoritmo coloca al elemento i en la posición adecuada para restablecer la propiedad.

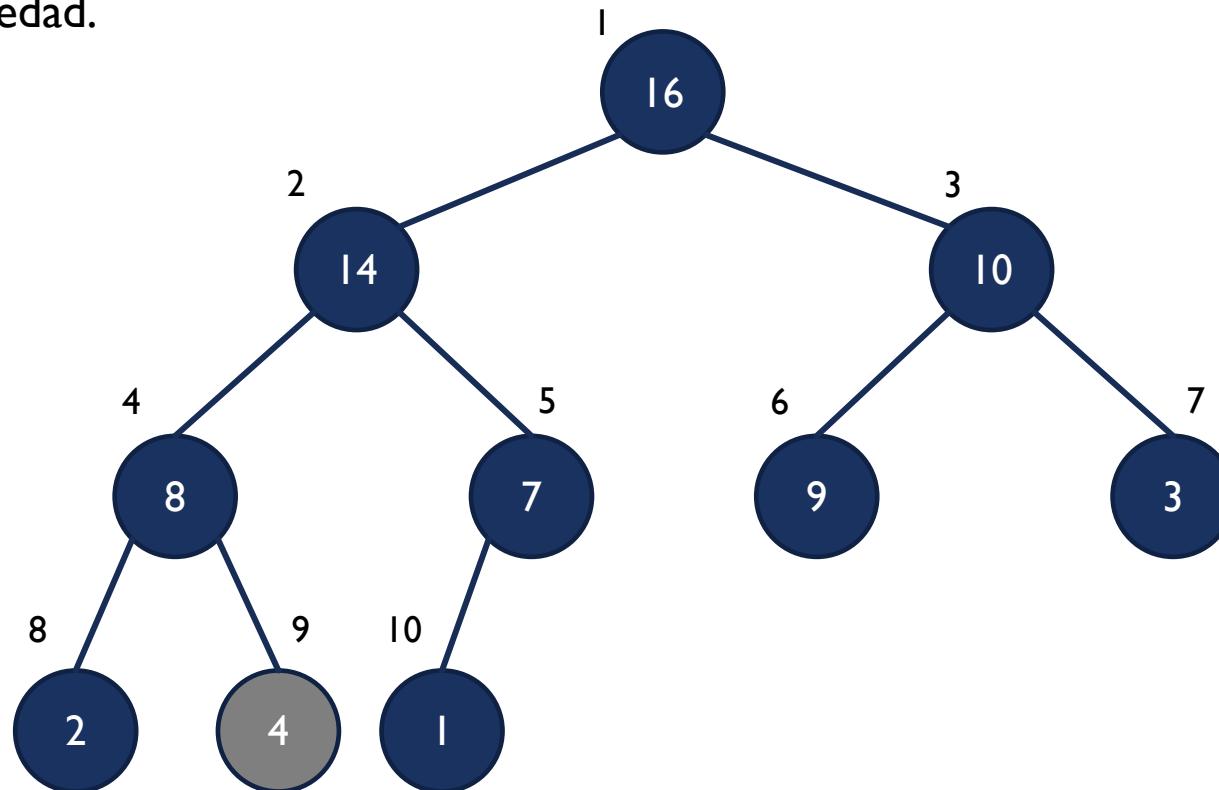
Suponiendo que se tiene el nodo 4 en la posición 2 del montículo, esta configuración viola la propiedad de montículo máximo:



Para restablecer la propiedad se debe intercambiar los nodos A[2] con A[4]:



Sin embargo, este movimiento rompe la propiedad en el nodo 4, por tanto, se debe hacer una llamada recursiva a la función hasta que se cumpla con la propiedad.



```
FUNC MAX-HEAPIFY(heap[ ]: INTEGER, id: INTEGER)
    largest ← id
    L ← LEFT(id)
    R ← RIGHT(id)
    SI L < length(heap) and heap[id] < heap[L]
        largest ← L
    SI R < length(heap) and heap[largest] < heap[R]
        largest ← R
    SI largest <> id
        swap heap[id] ←→ heap[largest]
        MAX-HEAPIFY(heap, largest)
    FIN_SI
FIN_FUNC
```

El algoritmo **MAX_HEAPIFY** utilizado con la estrategia BOTTOM-UP, permite convertir un conjunto de la forma A [l... n] en un montículo máximo.

La función **BUILD_MAX_HEAP** recorre todos los nodos del árbol (elementos del conjunto) y ejecuta el algoritmo **MAX_HEAPIFY** en cada iteración, para conservar la propiedad de montículo máximo.

```
FUNC BUILD-MAX-HEAP(heap[ ]: INTEGER)
    heapId ← length(heap)
    MIENTRAS heapId > - 1
        MAX-HEAPIFY(heap, heapId)
        heapId ← heapId - 1
    FIN_MIENTRAS
FIN_FUNC
```

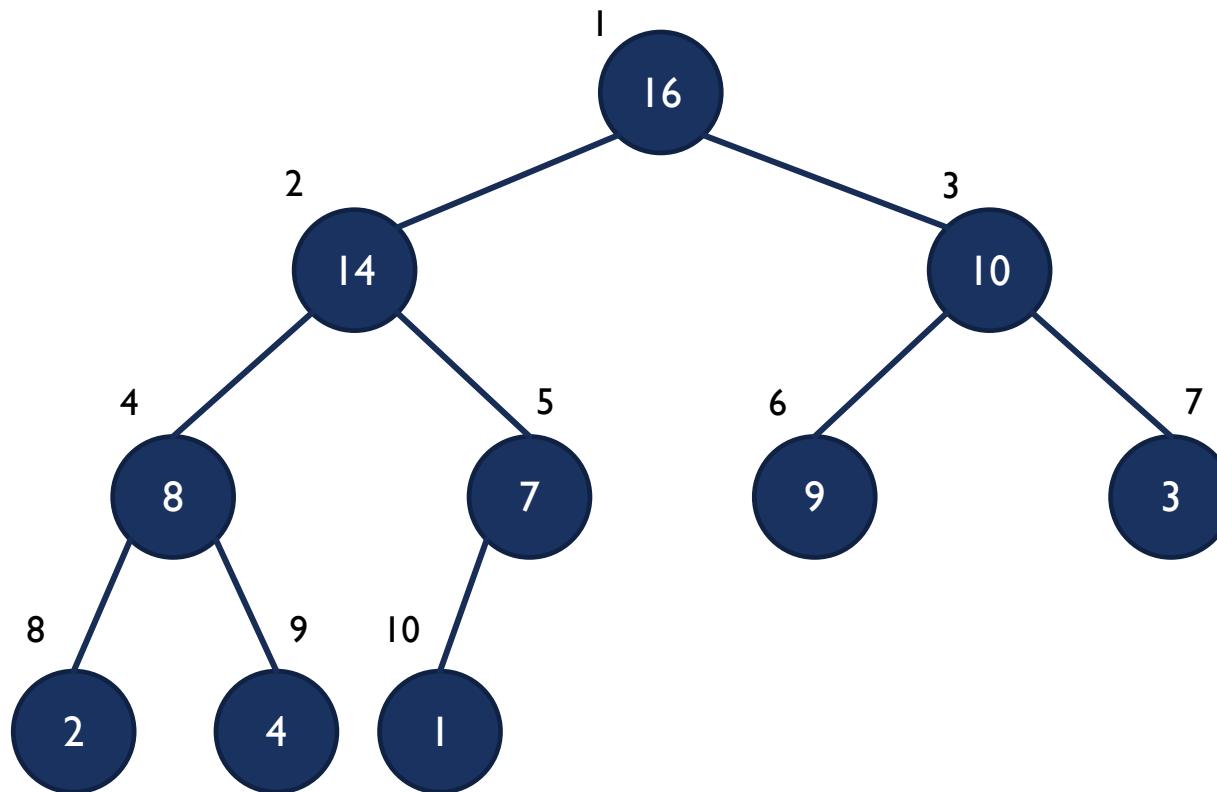
A partir de un montículo máximo, el algoritmo heap sort ordena los elementos. Debido a que la raíz de un montículo máximo contiene al elemento mayor del conjunto, éste puede ser almacenado en la posición correcta del conjunto ordenado.

En la siguiente iteración se coloca el último elemento del montículo en la raíz y se ejecuta la función MAX-HEAPIFY (con lo que se coloca el elemento mayor del conjunto restante de nuevo en la raíz). Este proceso se repite mientras existan elementos en el montículo.

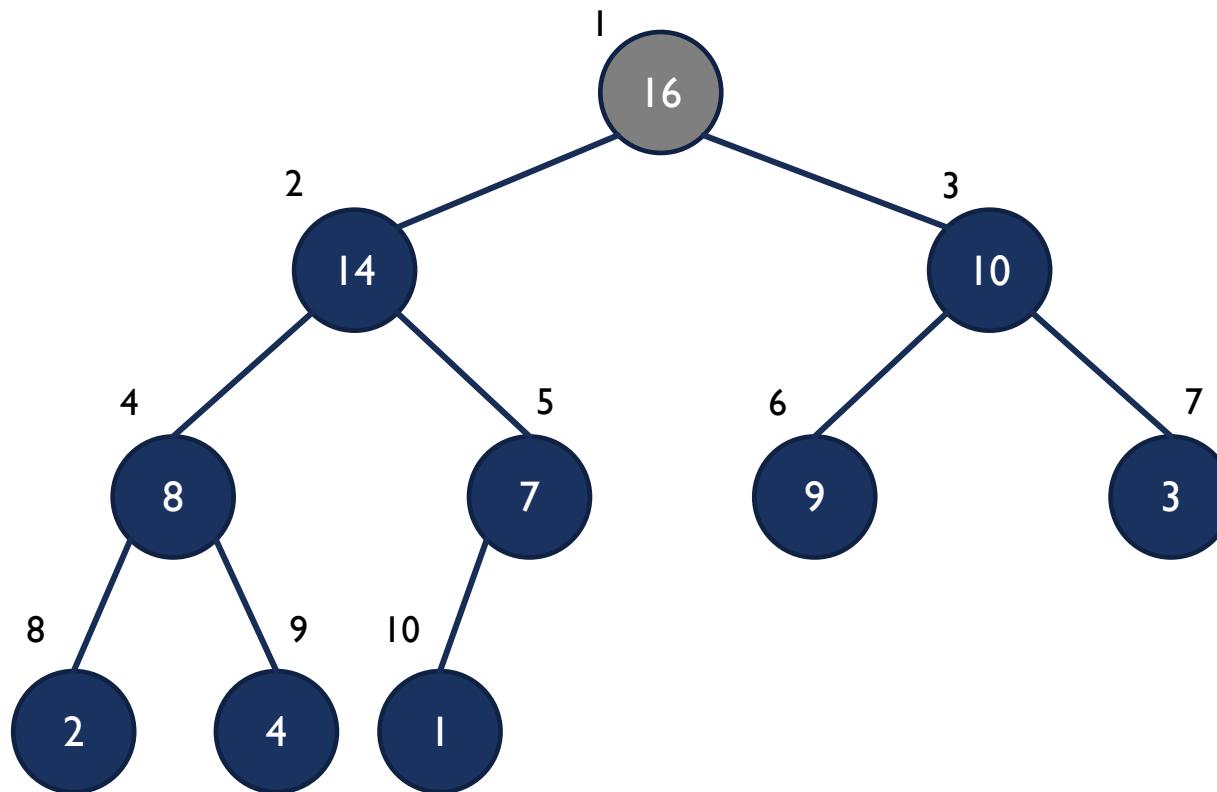
Ordenar los elementos del conjunto, de forma ascendente, utilizando heap sort.



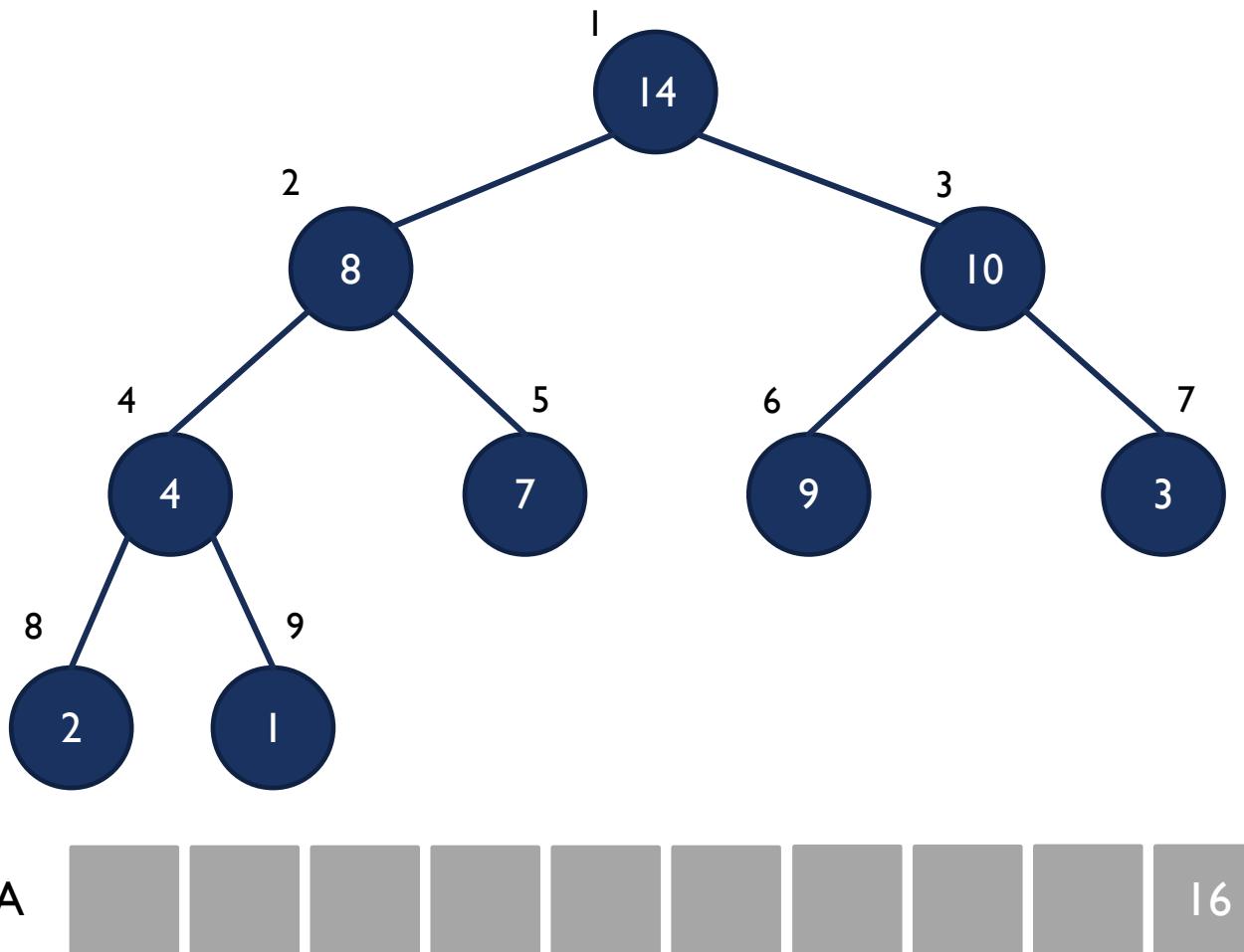
A partir del conjunto se genera un montículo máximo.



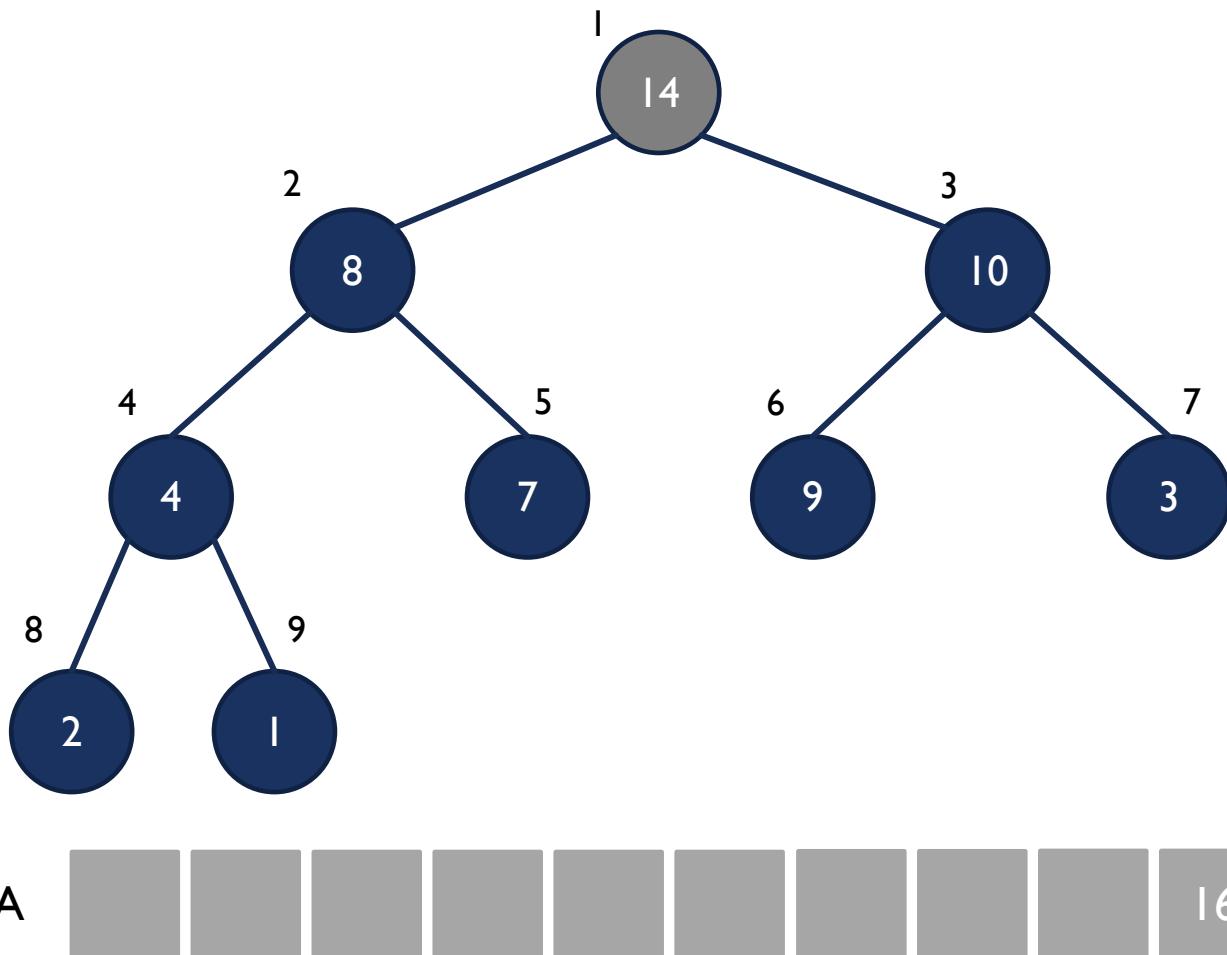
Se extrae la raíz del montículo y se coloca en la posición n del conjunto ordenado.



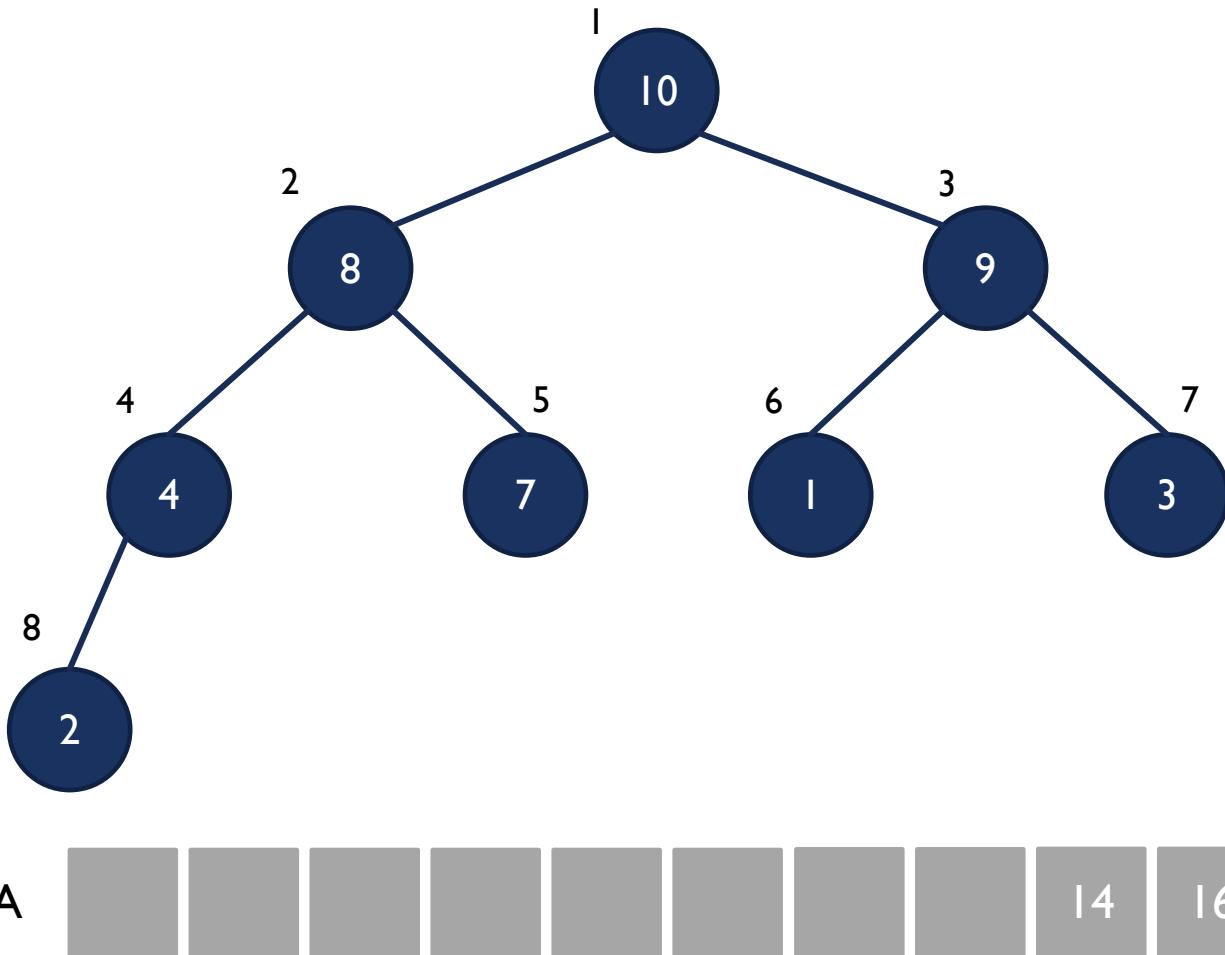
El hijo mayor pasa a ser la raíz mediante una llamada a la función MAX-HEAPIFY.



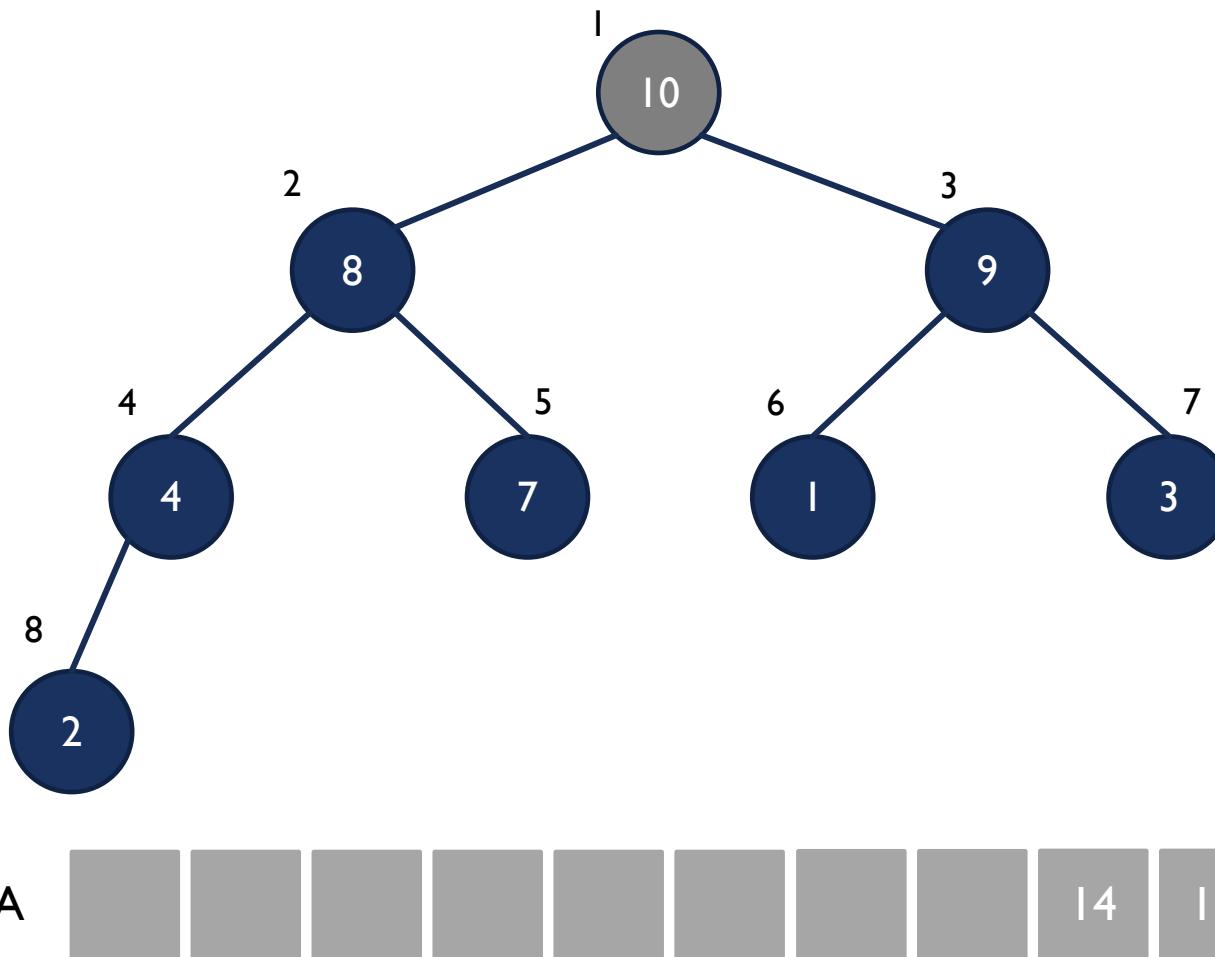
Se extrae la raíz del montículo y se coloca en la posición $n - 1$ del conjunto ordenado.



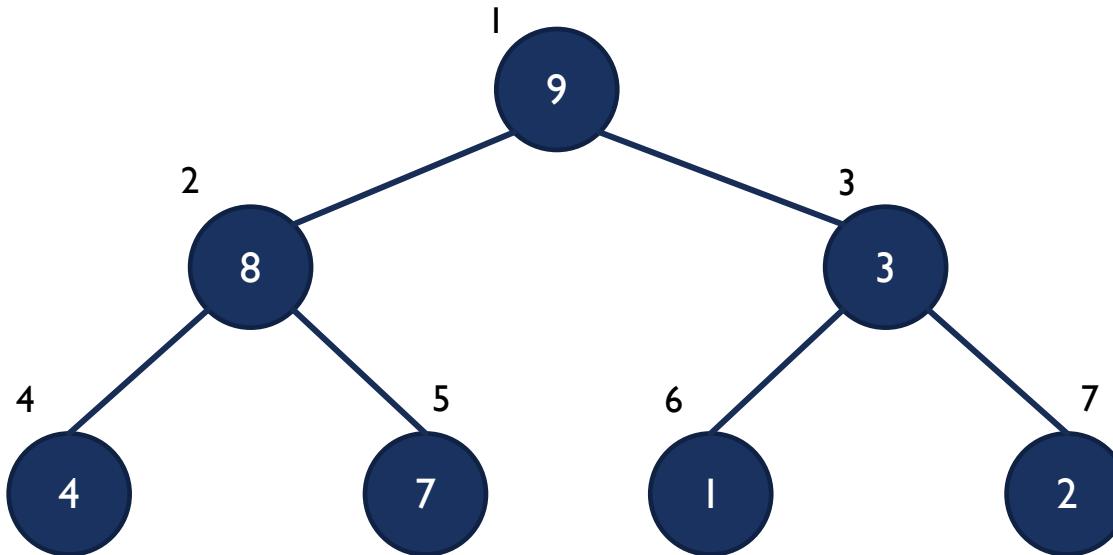
El hijo mayor pasa a ser la raíz mediante una llamada a la función MAXHEAPIFY.



Se extrae la raíz del montículo y se coloca en la posición $n - 2$ del conjunto ordenado.

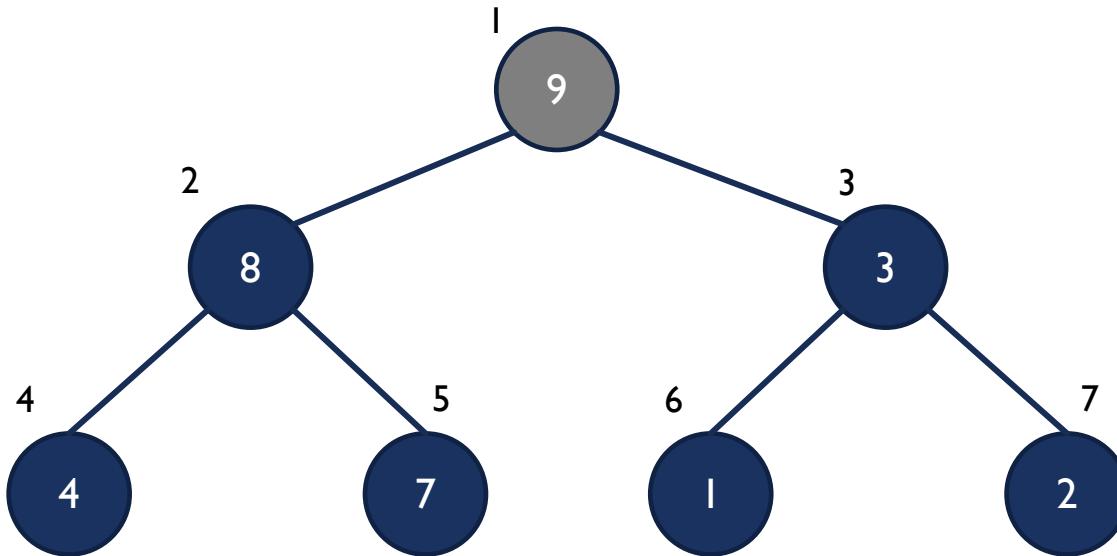


El hijo mayor pasa a ser la raíz mediante una llamada a la función MAX-HEAPIFY. El proceso se repite hasta que no haya elementos en el heap.



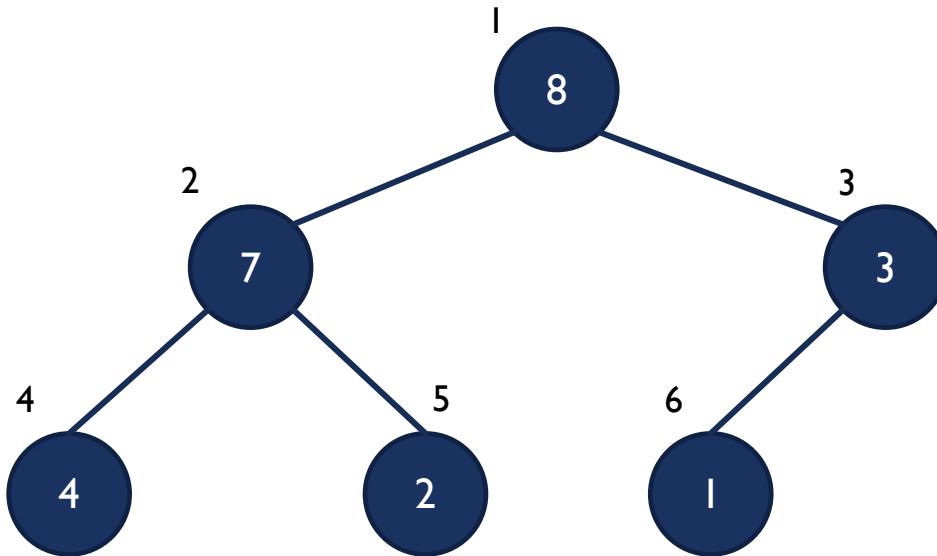
A





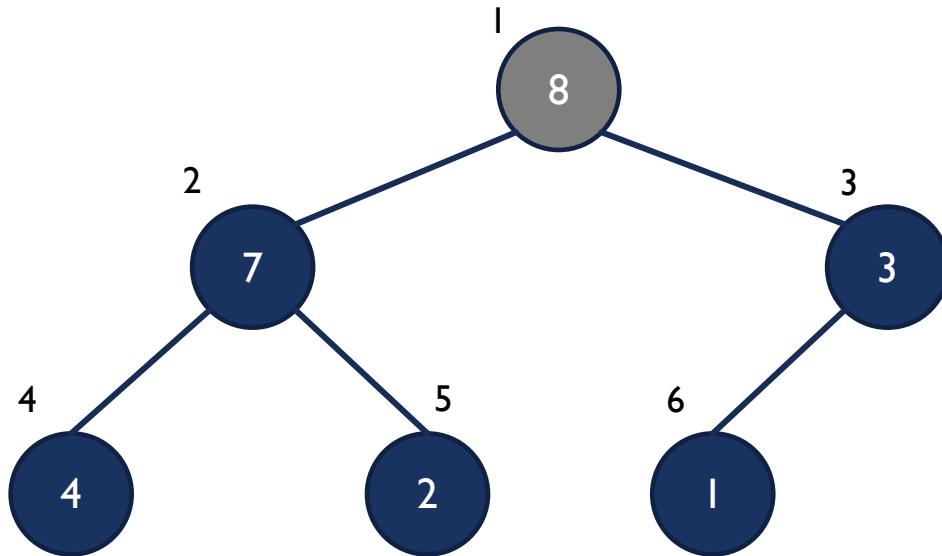
A





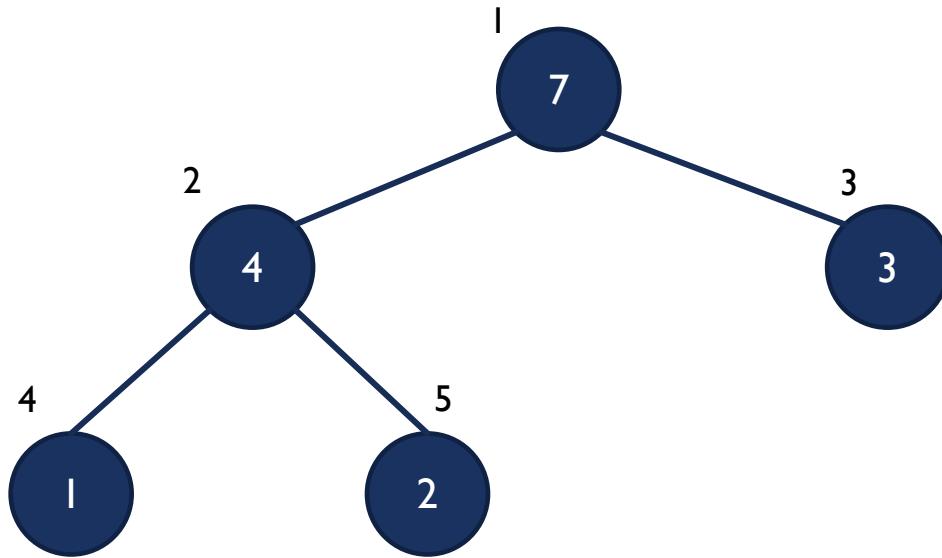
A





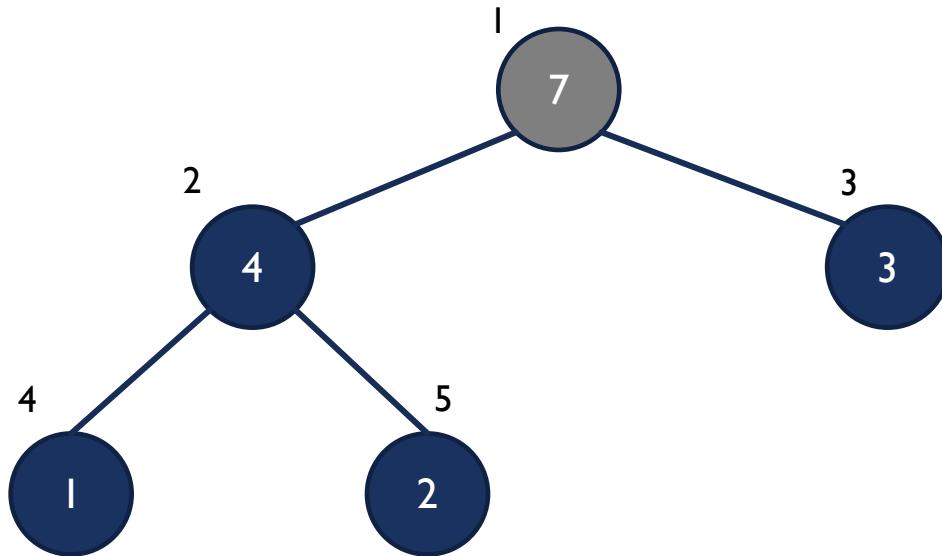
A





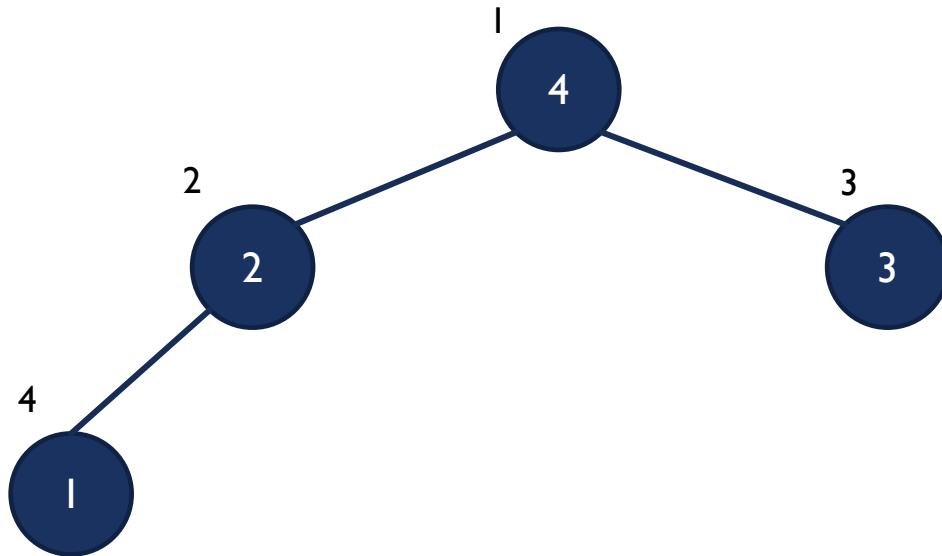
A





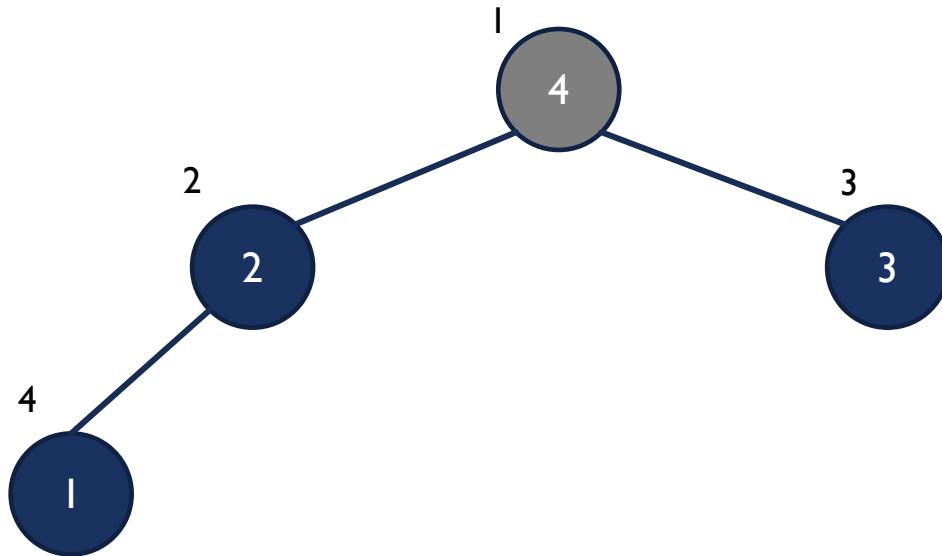
A





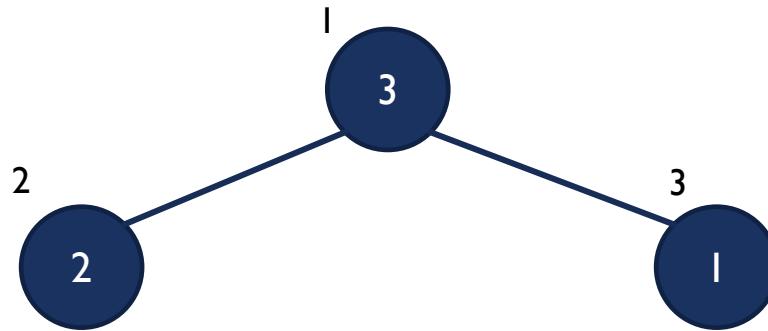
A





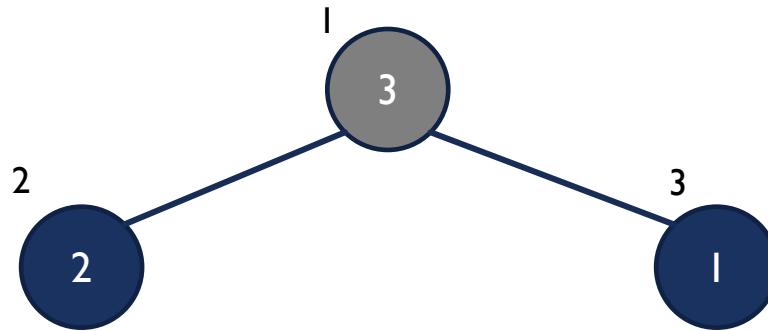
A





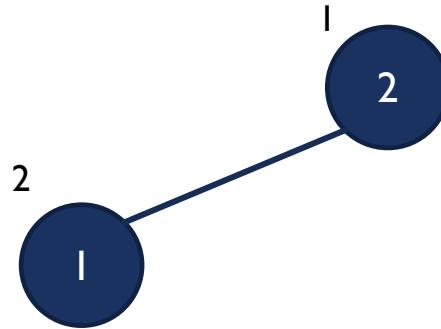
A





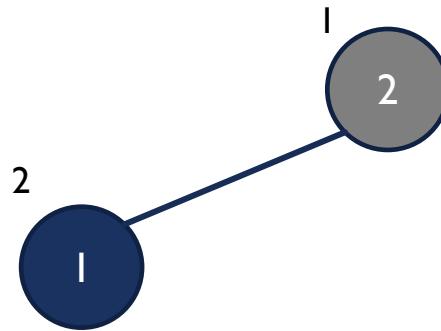
A





A





A





A

-
- 2
- 3
- 4
- 7
- 8
- 9
- 10
- 14
- 16



A

-
- 2
- 3
- 4
- 7
- 8
- 9
- 10
- 14
- 16



HEAP SORT

133



```
FUNC HEAP_SORT(heap[ ])
    BUILD-MAX-HEAP(heap)
    heapSize ← length(heap)
    MIENTRAS heapSize-1 >= 0
        swap heap[0] ↔ heap[heapSize]
        MAX-HEAPIFY(heap, heapSize)
        heapSize ← heapSize - 1
```



QUICK SORT

I.I.3 QUICK SORT

El método quick sort es uno de los más eficientes y veloces de los métodos de ordenamiento interno. También es conocido como método rápido o de ordenación por partición.

El algoritmo Quick Sort sigue los siguientes pasos:

- Se toma un elemento X de una posición cualquiera del conjunto.
- Se ubica al elemento X en la posición correcta del conjunto, de tal manera que todos los elementos que se encuentren a su izquierda sean menores o iguales a X y todos los elementos que se encuentren a su derecha sean mayores o iguales a X.
- Se repiten los pasos anteriores para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición de X en el conjunto.
- El proceso termina cuando todos los elementos se encuentran ordenados.

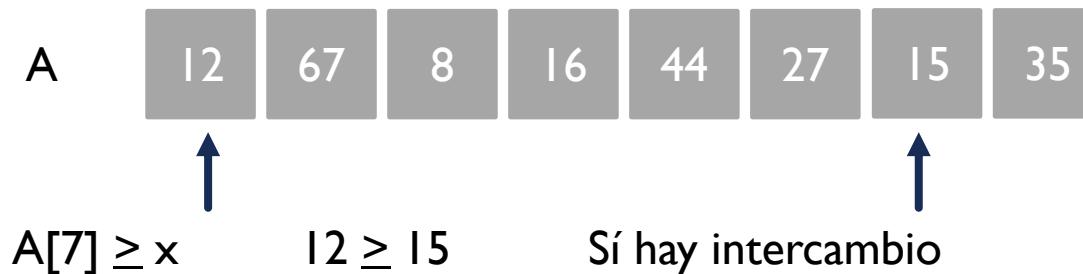
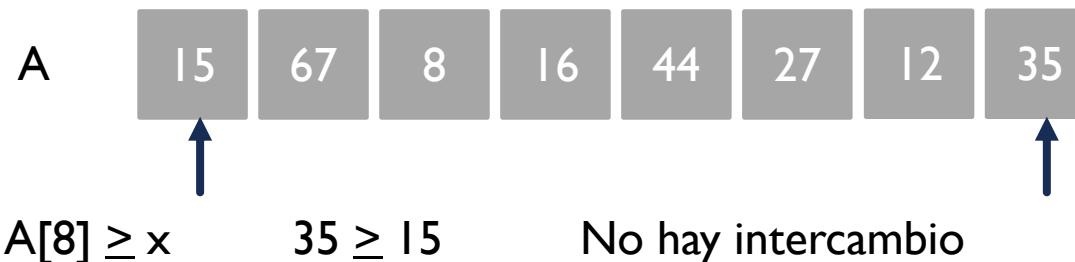
Ordenar el siguiente conjunto de forma ascendente utilizando el método de intercambio quick sort.

A

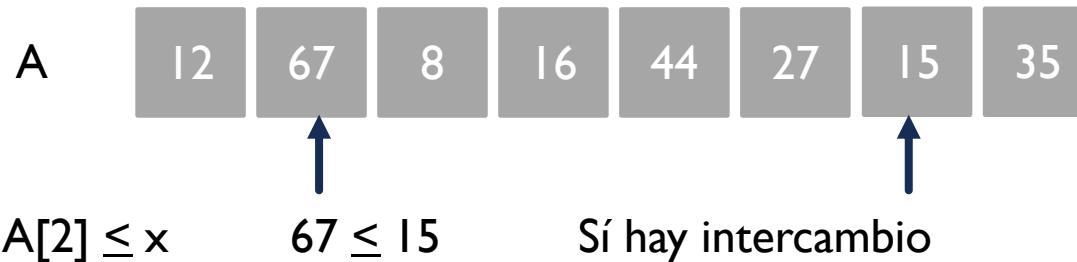


Se realiza el primer recorrido (de derecha a izquierdo).

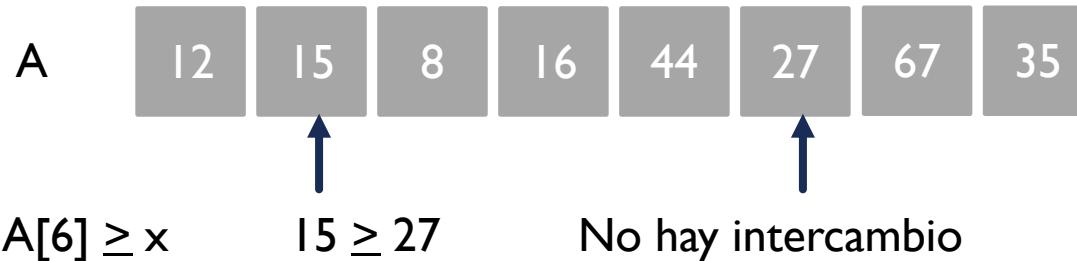
$$x \leftarrow A[1]$$



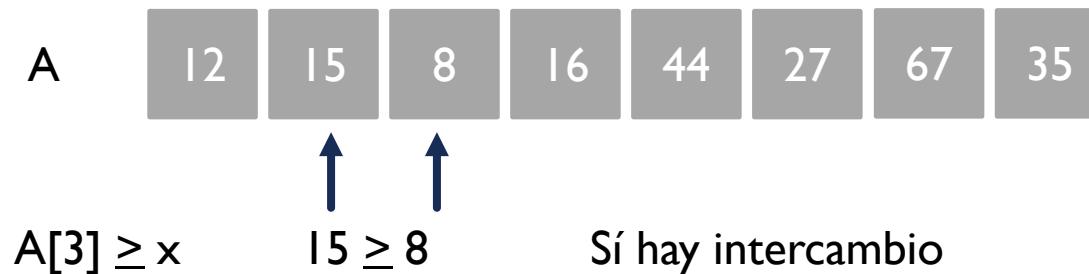
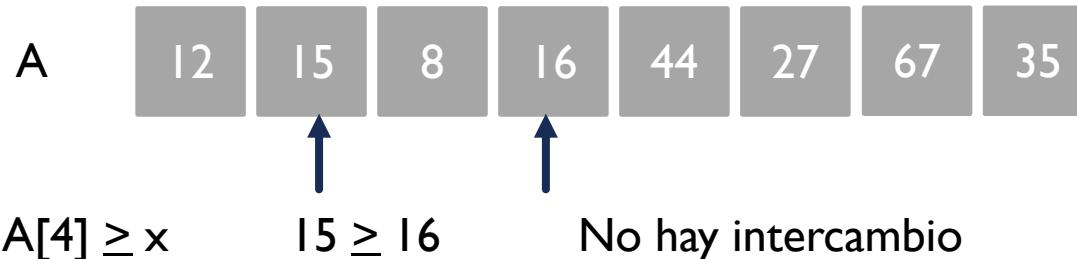
Se realiza el primer recorrido (de izquierda a derecha).

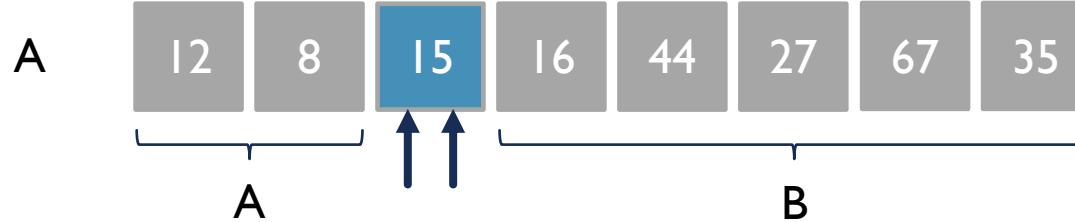


Se realiza el segundo recorrido (de derecha a izquierda).



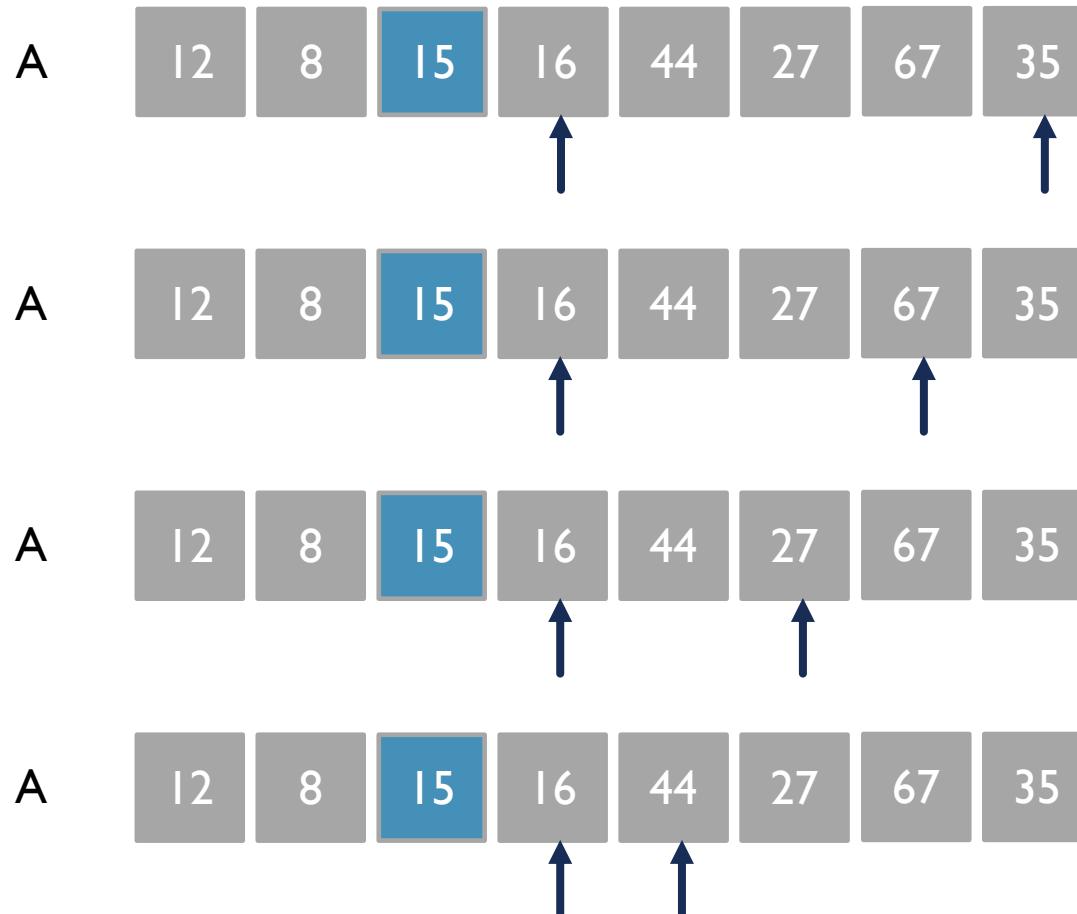
Se realiza el segundo recorrido (de derecha a izquierda).

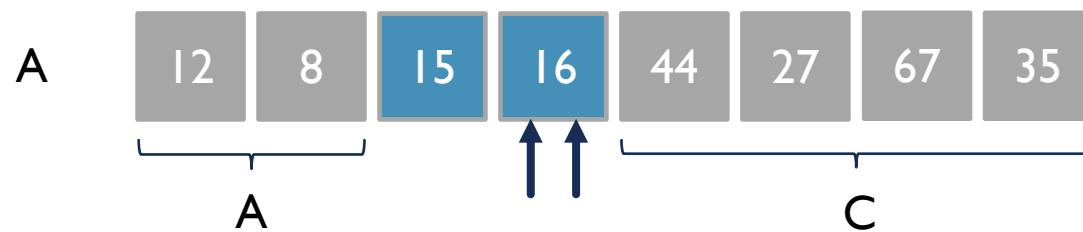


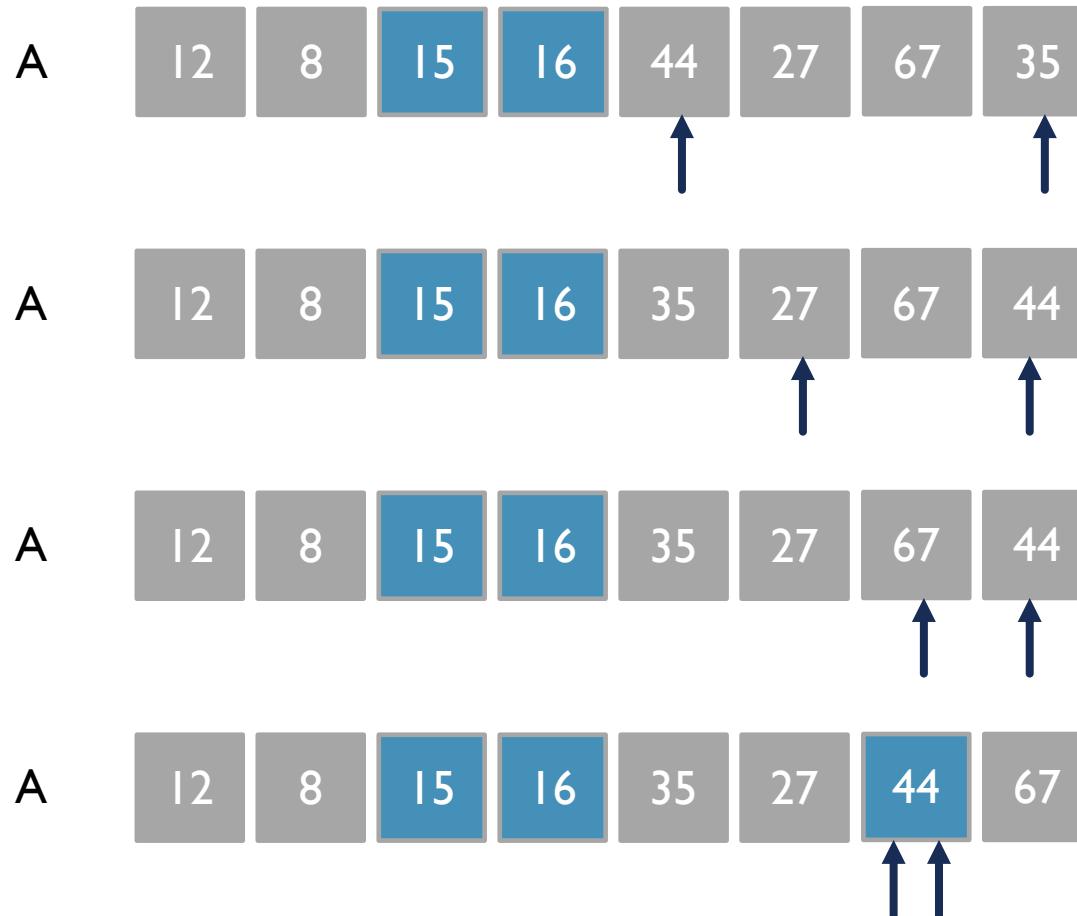


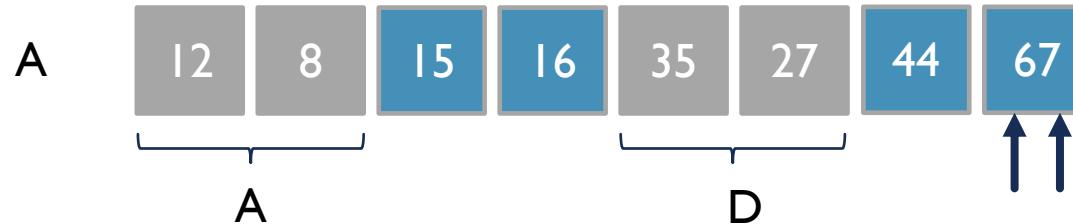
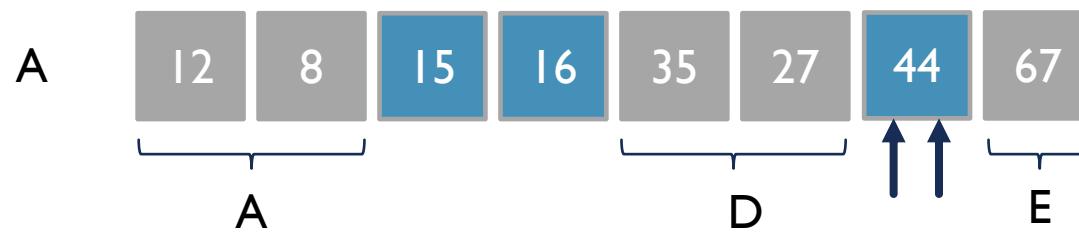
En este momento el proceso se detiene, ya que el número x elegido termina debido a que el recorrido de izquierda a derecha queda en la misma posición donde se encuentra el elemento x y, por tanto, x se encuentra en la posición correcta (los elementos del primer conjunto (conjunto izquierdo) son menores o iguales a x y los elementos del segundo conjunto (conjunto derecho) son mayores o iguales a x).

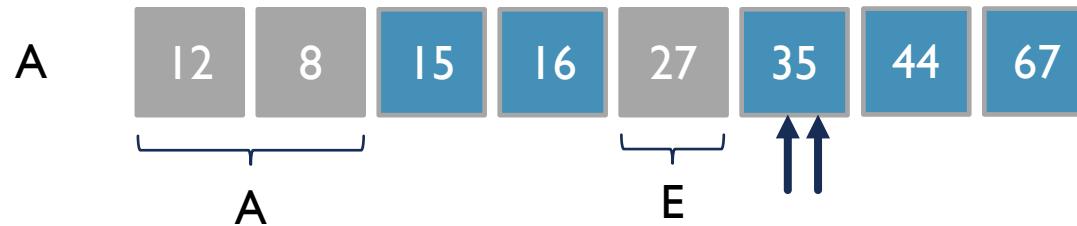
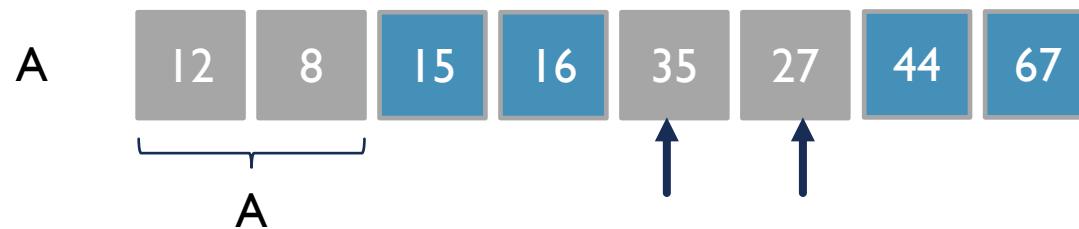
Ahora, se repite el proceso para cada uno de los conjuntos.

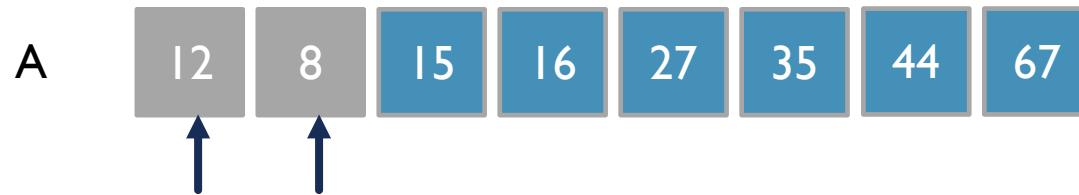
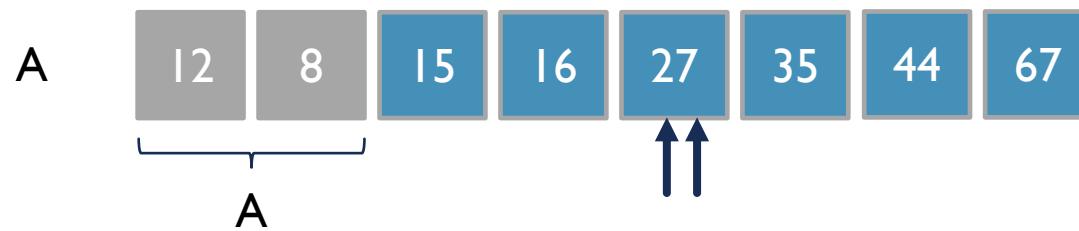










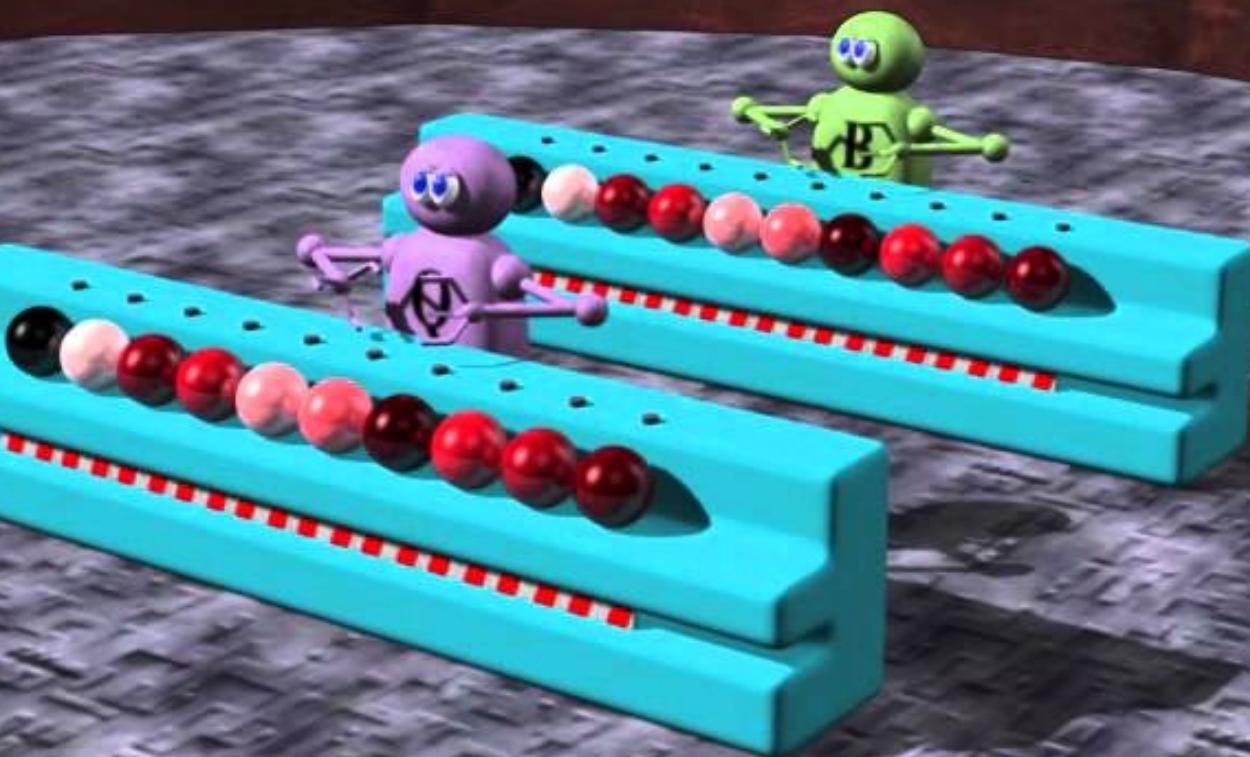




QUICK SORT

JRG SLN

151

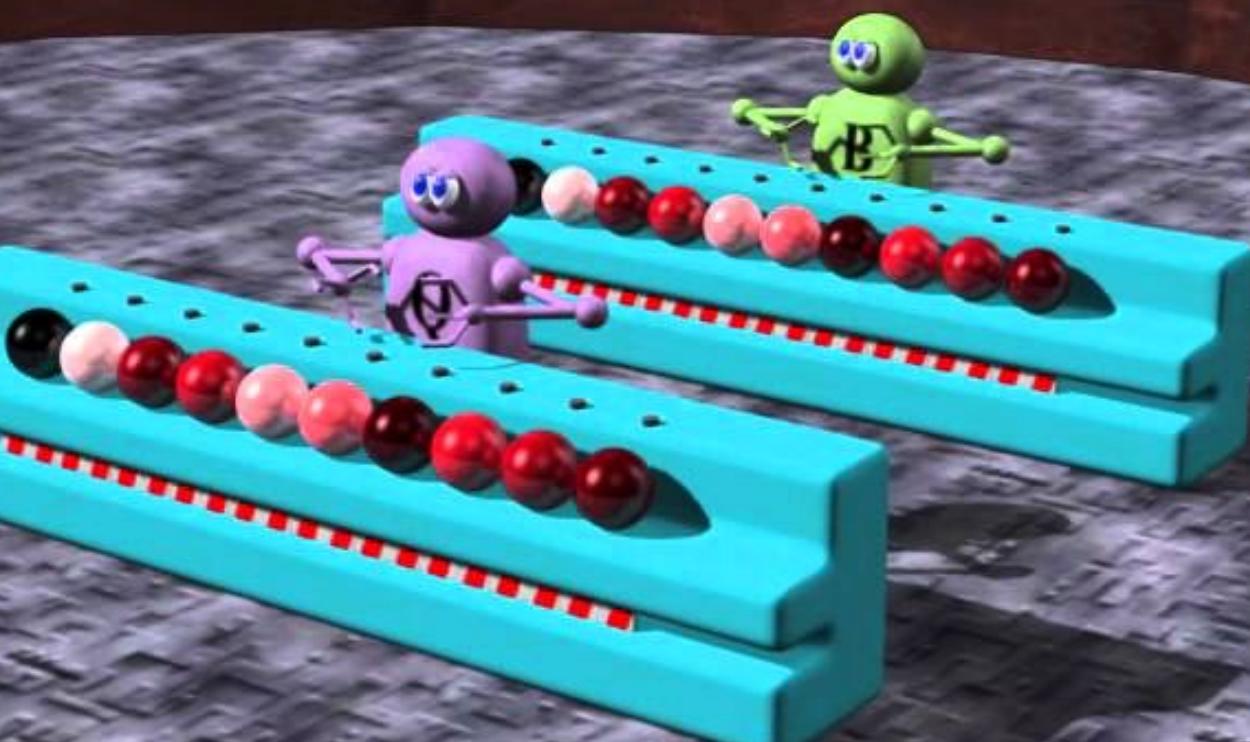


```
FUNC quick_sort(intList, intLow, intHigh):VACÍO
    SI intLow < intHigh ENTONCES
        pi ← partition(intList, intLow, intHigh)
        quick_sort(intList, intLow, pi-1)
        quick_sort(intList, pi+1, intHigh)
    FIN SI
FIN FUNC
```

```
FUNC partition(intList intLow, intHigh): ENTERO
    i ← intLow-1: ENTERO
    pivot ← intList [intHigh]
    j ← intLow: ENTERO
    MIENTRAS j < intHigh ENTONCES
        SI intList [j] <= pivot ENTONCES
            i ← i+1
            intList[i] ↔ intList[j]
        FIN SI
        j ← j + 1
    FIN MIENTRAS
    intList[i+1] ↔ intList[intHigh]
    dev (i+1)
FIN FUNC
```

QUICK SORT VS MERGE SORT

154





ALGORITMOS DE ORDENAMIENTO. PARTE 2.

PRÁCTICA 2

ALGORITMOS DE ORDENAMIENTO. PARTE 2.

- Implementar los algoritmos de Heap sort y Quick sort en Python.
- Obtener los polinomios del mejor, el peor y el caso promedio de complejidad de cada algoritmo (Quick sort y Heap sort).
- Graficar en Python el comportamiento de los algoritmos implementados para diferentes instancias de tiempo (listas de 1 a 1000 elementos) para el mejor (lista ordenada), el peor (lista ordenada en forma inversa) y el caso promedio (lista aleatoria) de complejidad.



COUNTING SORT

I.I.4 COUNTING SORT

El algoritmo counting sort se basa en determinar para cada elemento x del conjunto, el número de elementos menores a x . Con esa información, es posible poner el elemento x en su posición correcta dentro del conjunto ordenado.

El algoritmo counting sort asume que los elementos que conforman el conjunto son enteros positivos (incluyendo a cero).

El algoritmo utiliza 3 conjuntos de datos A, B y C. El conjunto A es el conjunto de datos de entrada. El conjunto C es el que determina la posición correcta de cada elemento. El conjunto B es el conjunto con el arreglo ordenado.

Para definir el tamaño de C se debe extraer el valor máximo de A ($\max(A)$). C es de tamaño $[0 \dots \max(A)]$. B es del mismo tamaño de A, es decir, $B[1 \dots \text{length}(A)]$.



El algoritmo se basa en contabilizar el número de veces que se repite un número en el conjunto de datos. Este conteo se almacena en el conjunto C.

Una vez que se posee el número de repeticiones de cada elemento en el conjunto es posible establecer la posición final de cualquier elemento.

Ordenar el siguiente conjunto de forma ascendente utilizando el método de counting sort.

A



Primero se debe obtener el máximo del conjunto A para poder determinar el tamaño del conjunto C, es decir, $\text{maximum}(A)$.

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

El tamaño del conjunto C está determinado por el número menor del conjunto (recordar que counting sort asume que el conjunto A es un conjunto de datos enteros positivos, incluyendo a cero).

	0	1	2	3	4	5
C	0	0	0	0	0	0

Los elementos ordenados se guardarán en el conjunto B, el cual es del mismo tamaño de A, es decir, $B[1 \dots \text{length}(A)]$.



Una vez creados los 3 conjuntos, se procede a contabilizar el número de veces que se repite cada elemento del conjunto A y se guarda la información en el conjunto C.

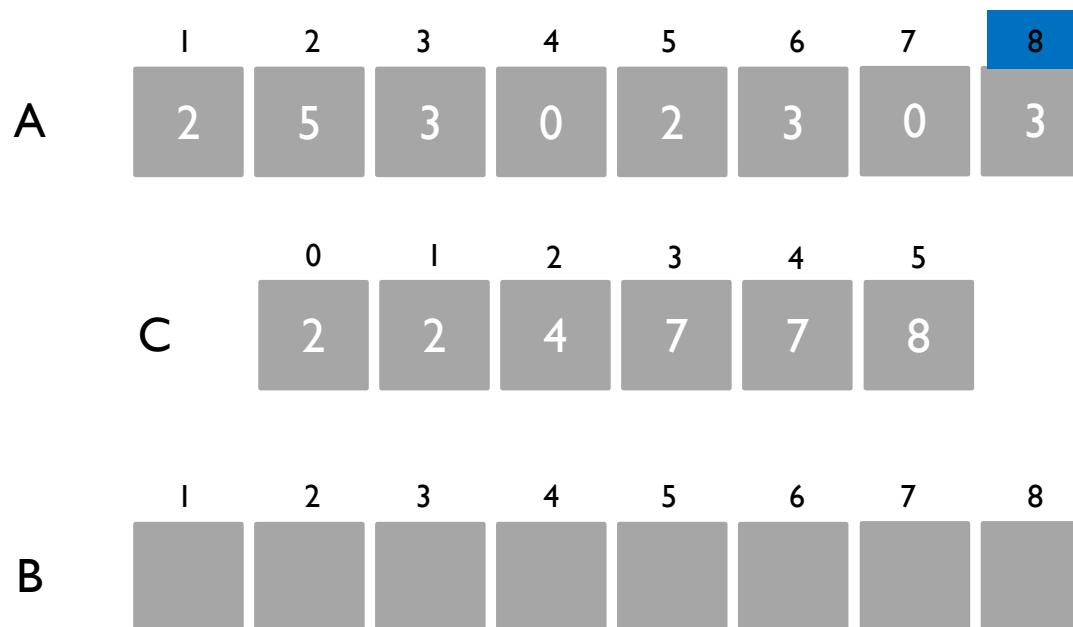
	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
C	0	1	2	3	4	5		
	2	0	2	3	0	1		

Para determinar la posición correcta de cada elemento, es necesario sumar las repeticiones de todos los elementos anteriores, por lo tanto, dicha suma se realiza dentro del conjunto C

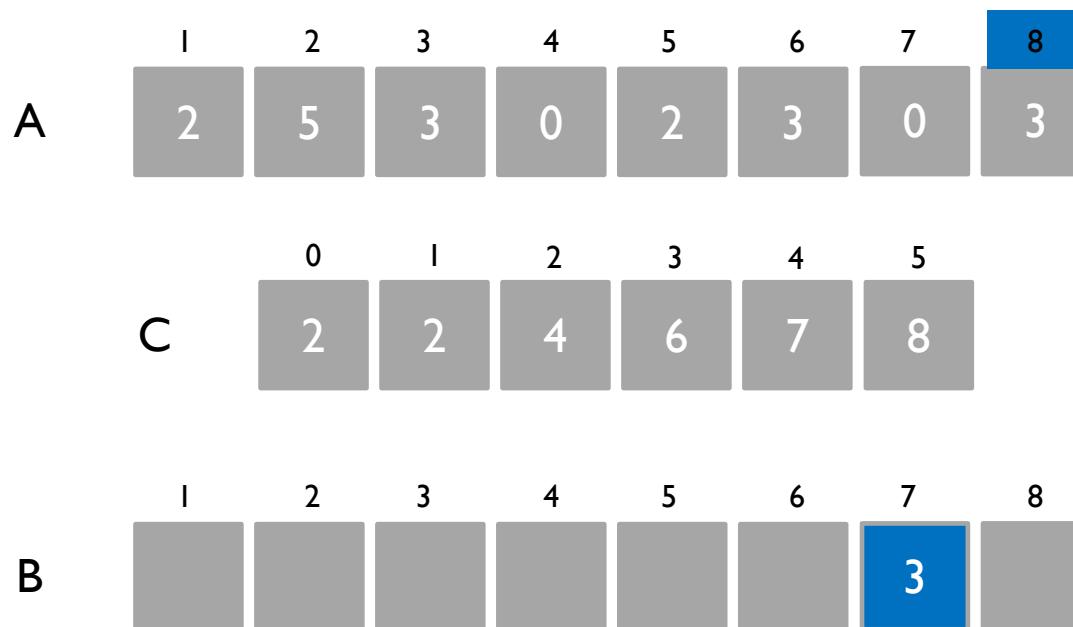
	0	1	2	3	4	5
C	2	0	2	3	0	1

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

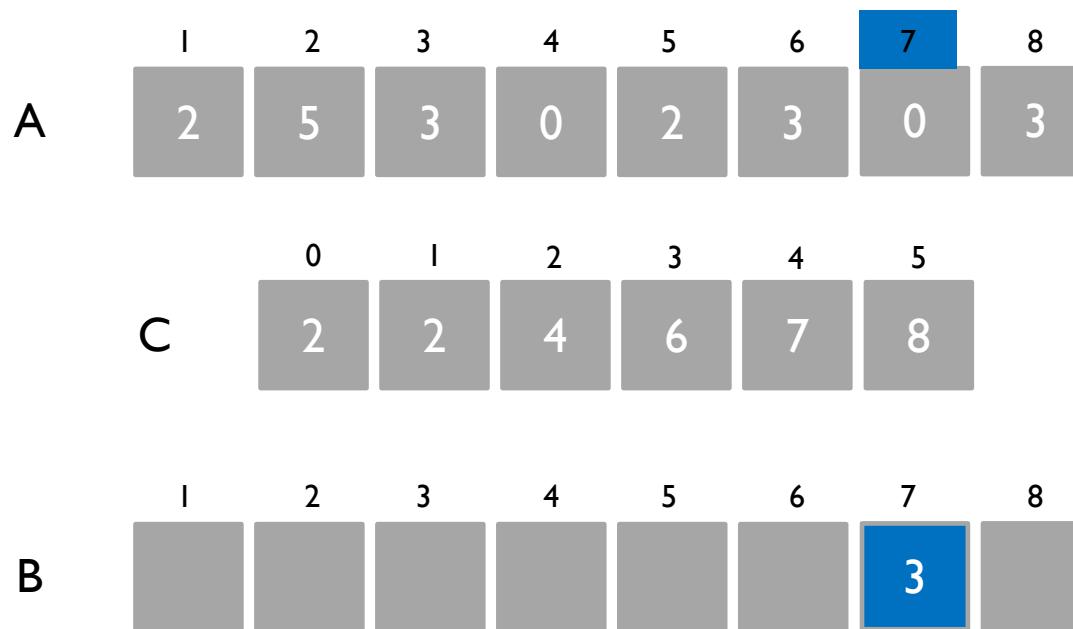
Se accede al elemento enésimo del conjunto A y se establece en el conjunto B dependiendo de la posición de C, es decir, $B[C[A[n]]] = A[n]$, y se decrementa $C[A[n]]$.



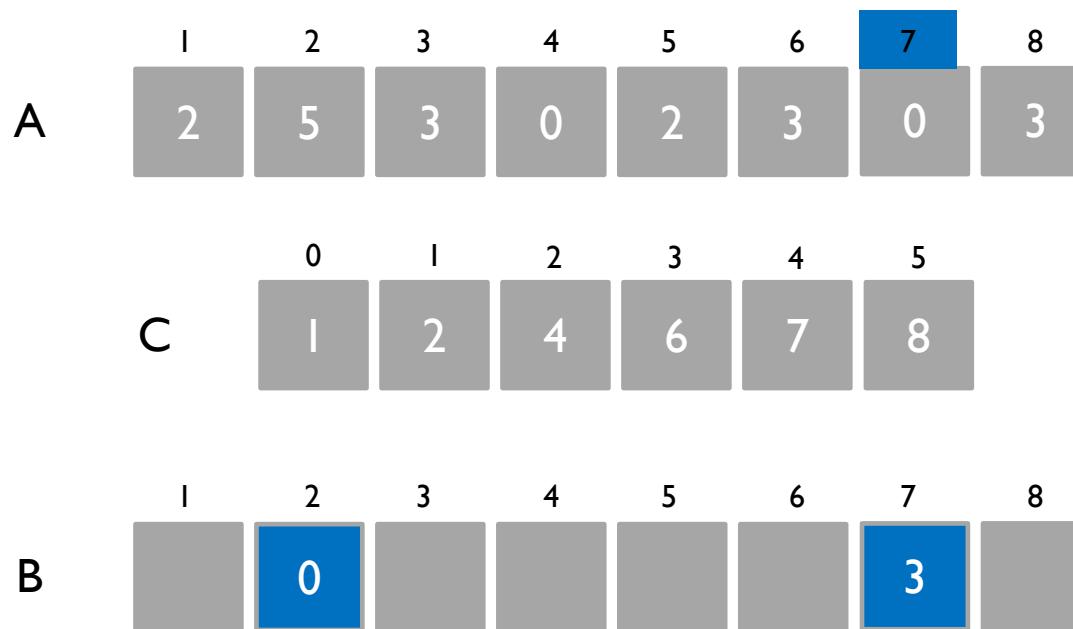
Se accede al elemento enésimo del conjunto A y se establece en el conjunto B dependiendo de la posición de C, es decir, $B[C[A[n]]] = A[n]$, y se decrementa $C[A[n]]$.



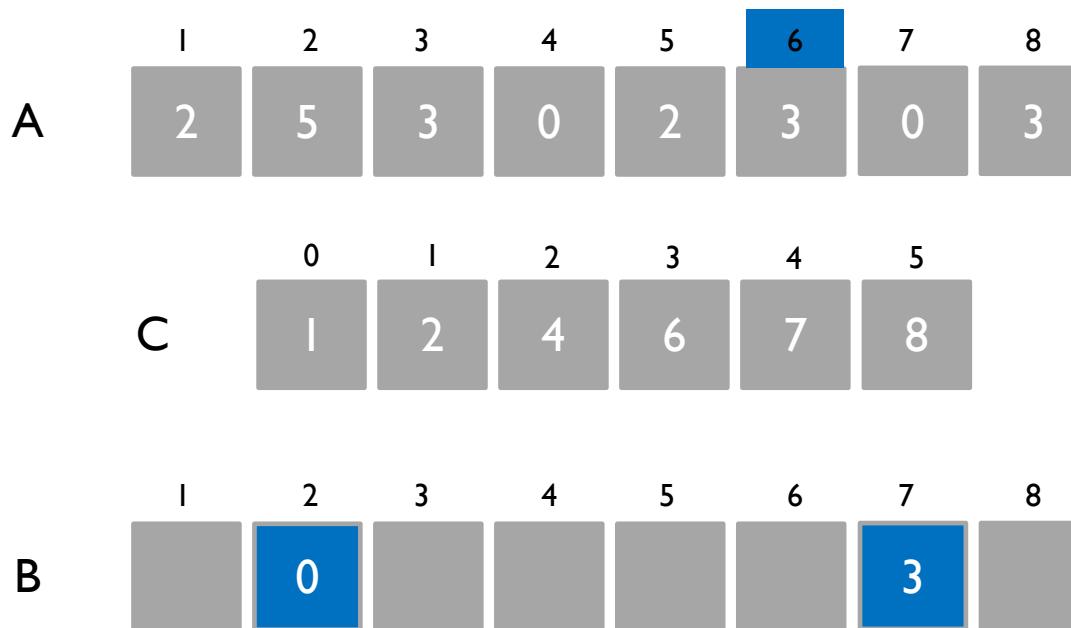
Se accede al elemento enésimo - I del conjunto A y se establece en el conjunto B dependiendo de la posición de C, es decir, $B[C[A[n-I]]] = A[n-I]$, y se decrementa C[A[n-I]].



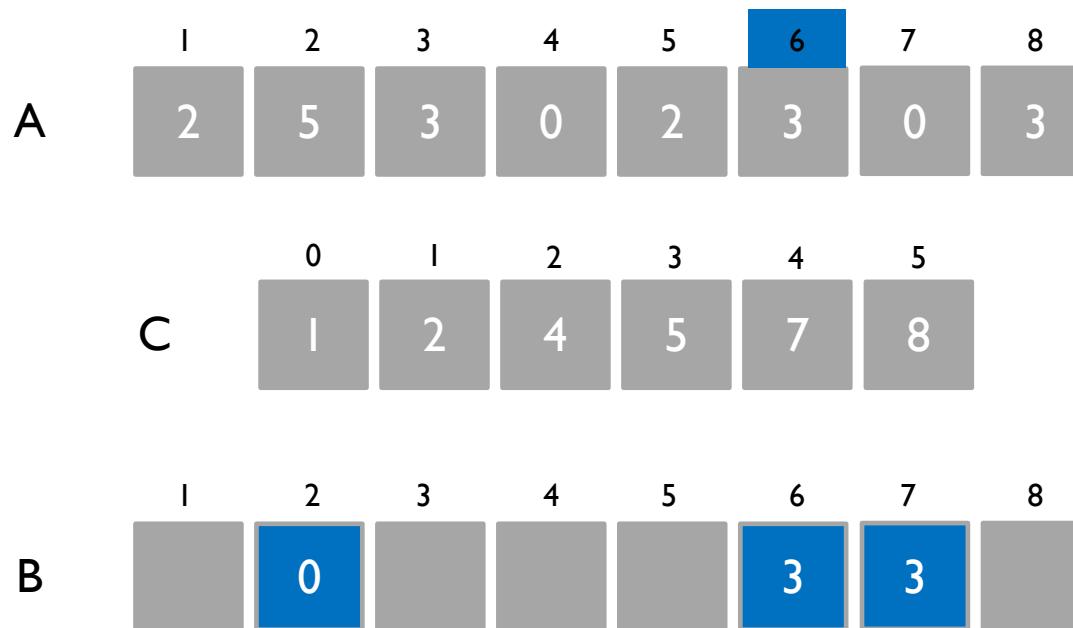
Se accede al elemento enésimo - I del conjunto A y se establece en el conjunto B dependiendo de la posición de C, es decir, $B[C[A[n-I]]] = A[n-I]$, y se decrementa C[A[n-I]].



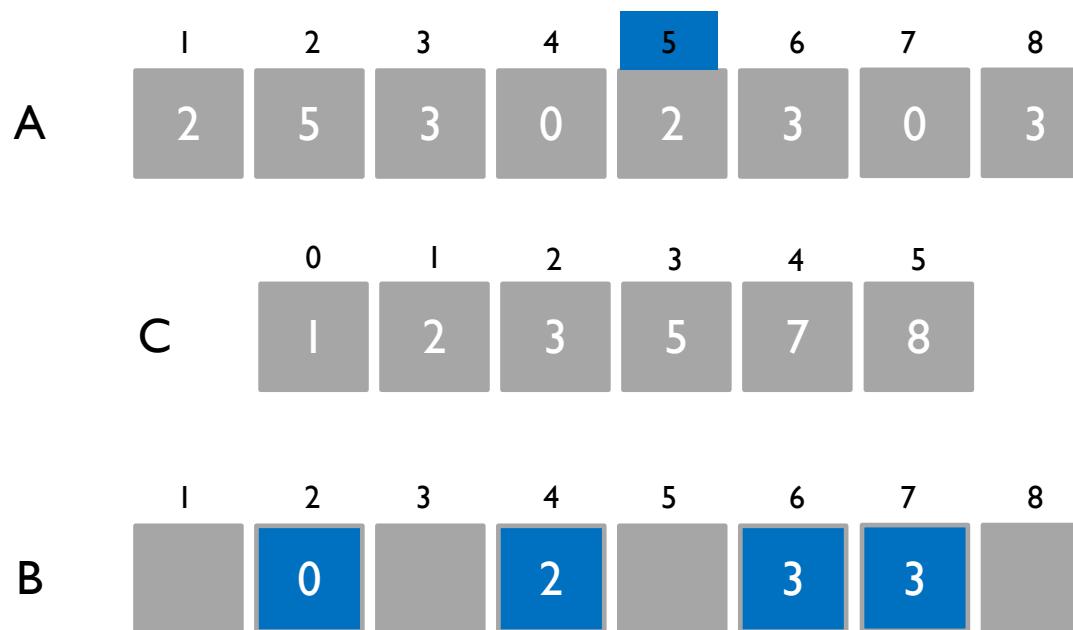
El proceso se repite hasta llegar al primer elemento del conjunto A.



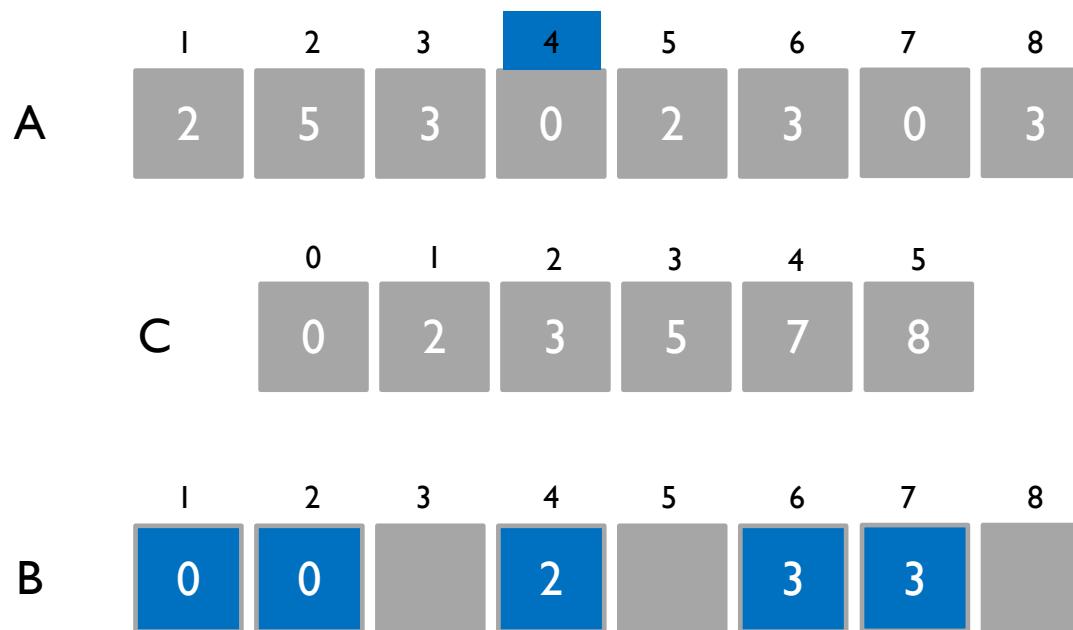
El proceso se repite hasta llegar al primer elemento del conjunto A.



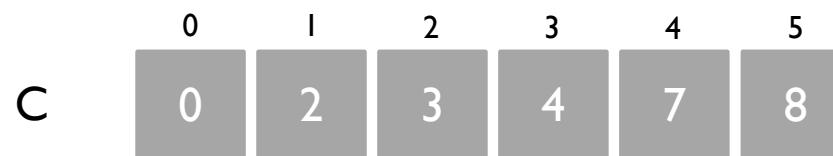
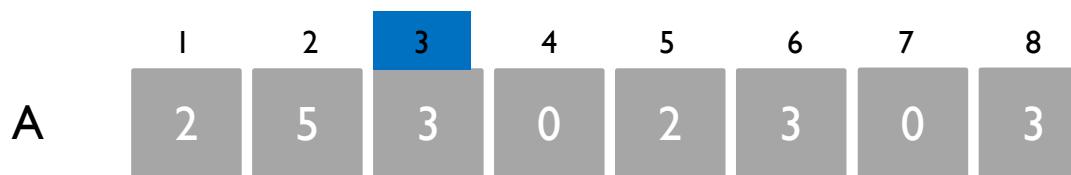
El proceso se repite hasta llegar al primer elemento del conjunto A.



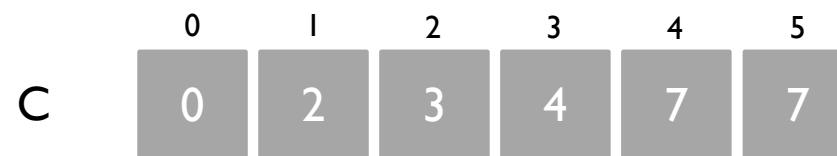
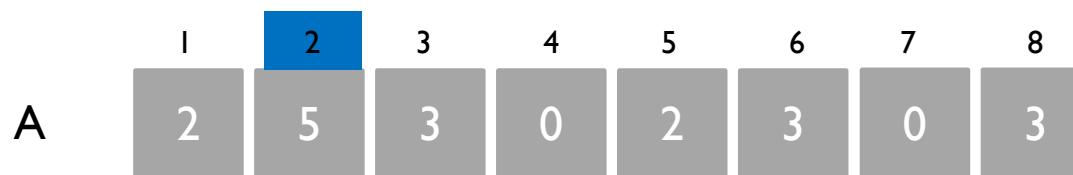
El proceso se repite hasta llegar al primer elemento del conjunto A.



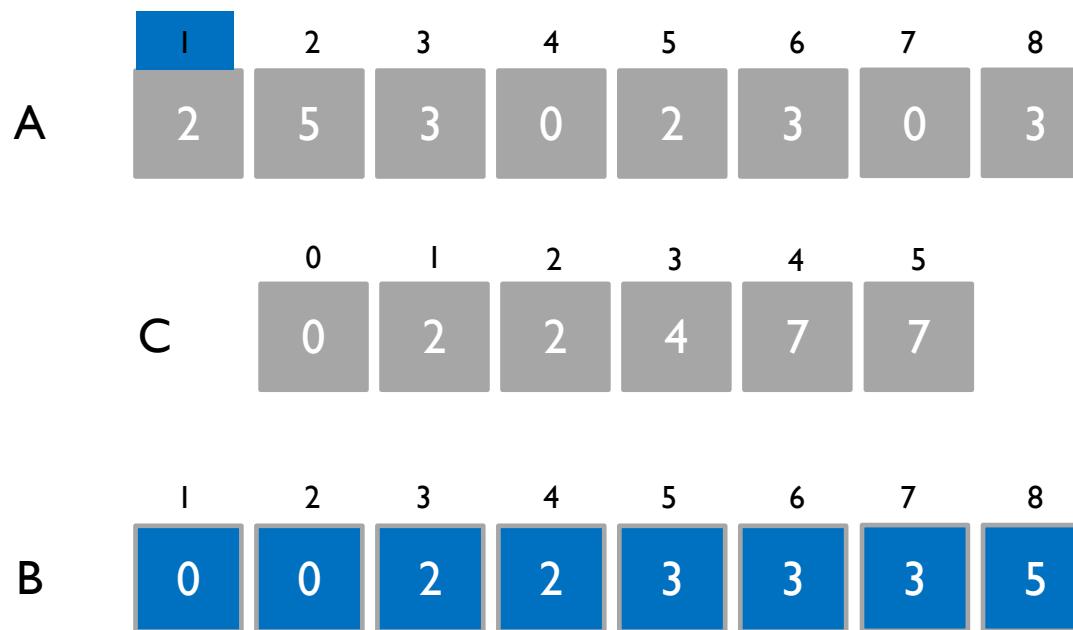
El proceso se repite hasta llegar al primer elemento del conjunto A.



El proceso se repite hasta llegar al primer elemento del conjunto A.

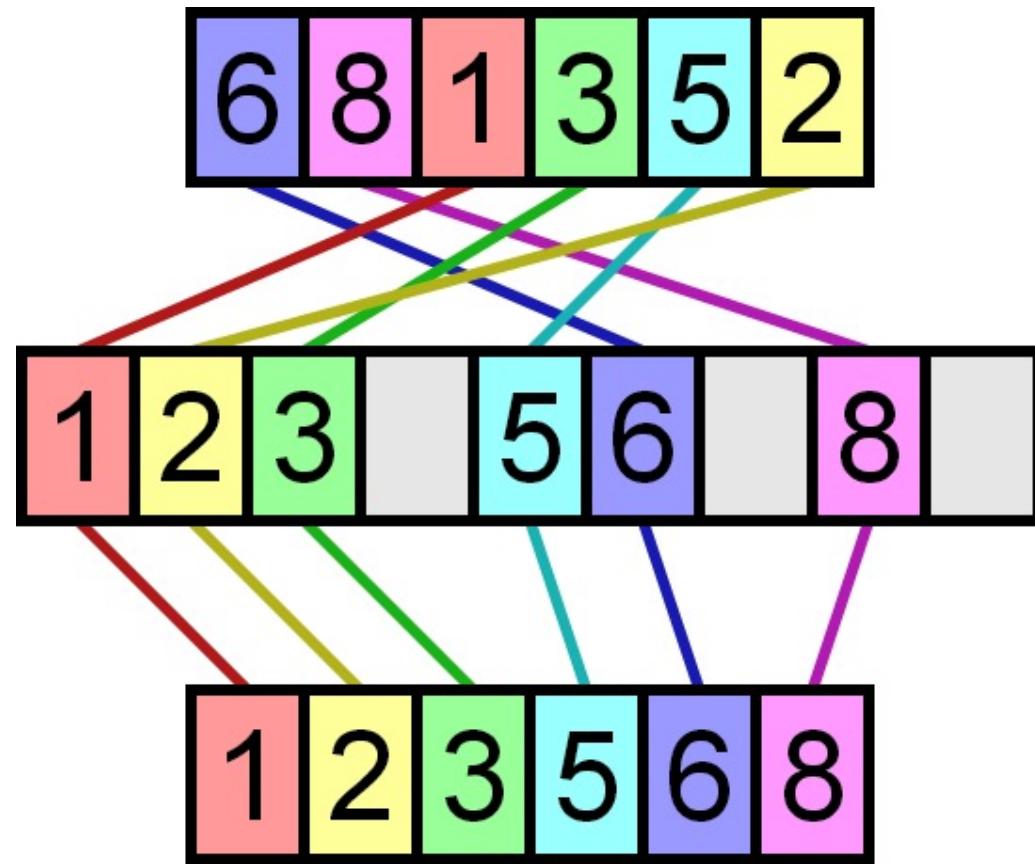


El proceso se repite hasta llegar al primer elemento del conjunto A.



Al finalizar el recorrido del conjunto A, la secuencia de números ordenada se encuentra en el conjunto B.

B	1	2	3	4	5	6	7	8
	0	0	2	2	3	3	3	5



COUNTING SORT

```
FUNC counting_sort(A): LISTA
    i ← 0: ENTERO
    MIENTRAS i < length(A) ENTONCES
        C[ordinary(A[i])] ← C[ordinary(A[i])] + 1
        i ← i + 1
    FIN MIENTRAS
    i ← 1
    MIENTRAS i < length(C) ENTONCES
        C[i] ← C[i] + C[i-1]
        i ← i + 1
    FIN MIENTRAS
    i = 0
    MIENTRAS i < length(A) ENTONCES
        B[C[ord(A[i])-1]] = A[i]
        C[ord(A[i])] ← C[ord(A[i])] - 1
        i ← i + 1
    FIN MIENTRAS
    DEV B
FIN FUNC
```



RADIX SORT

I.I.5 RADIX SORT

El ordenamiento radix (Radix Sort) es el más representativo de los métodos de distribución. Es un algoritmo de ordenamiento estable que puede ser usado para ordenar elementos identificados por llaves (o claves) únicas. Cada llave debe ser una cadena o un número capaz de ser ordenada alfanuméricamente.

Este método también es conocido como *Clasificación por urnas*, debido a que realiza el almacenamiento de datos en cajas para construir conjuntos de elementos.

Los pasos que sigue radix sort son:

- Recorrer todos los elementos del conjunto empezando por el dígito menos significativo (el bit más a la derecha).
- Hacer coincidir el dígito leído con los dígitos correspondientes en las urnas o casillas.
- Extraer los elementos de las casillas y repetir el proceso con el siguiente dígito hasta llegar al tamaño máximo del número de dígitos.

Ordenar el siguiente conjunto de forma ascendente utilizando el método de distribución Radix Sort.

A

31

41

81

54

23

33

64

98

93

88

Se realizan diversos montones de elementos. El número máximo de elementos está dado por el tamaño del conjunto. Cada elemento se lee dígito a dígito, de derecha a izquierda, iniciando por el dígito menos significativo (0).

A	31	41	81	54	23	33	64	98	93	88
---	----	----	----	----	----	----	----	----	----	----

0	1	2	3	4	5	6	7	8	9
	31		23	54				98	
	41		33	64				88	
	81		93						

Después se extraen los montones en orden, es decir, de la columna uno se extraen todos los elementos de forma FIFO, de la columna dos se extraen todos los elementos de forma FIFO y así hasta la columna n.

0	1	2	3	4	5	6	7	8	9
	31		23	54				98	
	41		33	64				88	
	81		93						



Se vuelven a realizar montones de elementos pero ahora con el dígito siguiente (1).



0	1	2	3	4	5	6	7	8	9
		23	31	41	54	64		81	93
			33					88	98

Se vuelven a extraer los montones en orden de forma FIFO de la columna 1 a la columna n.

0	1	2	3	4	5	6	7	8	9
		23	31	41	54	64		81	93
			33					88	98



Cuando se llega al dígito más significativo de todos los números, el proceso termina y el conjunto ya está ordenado.



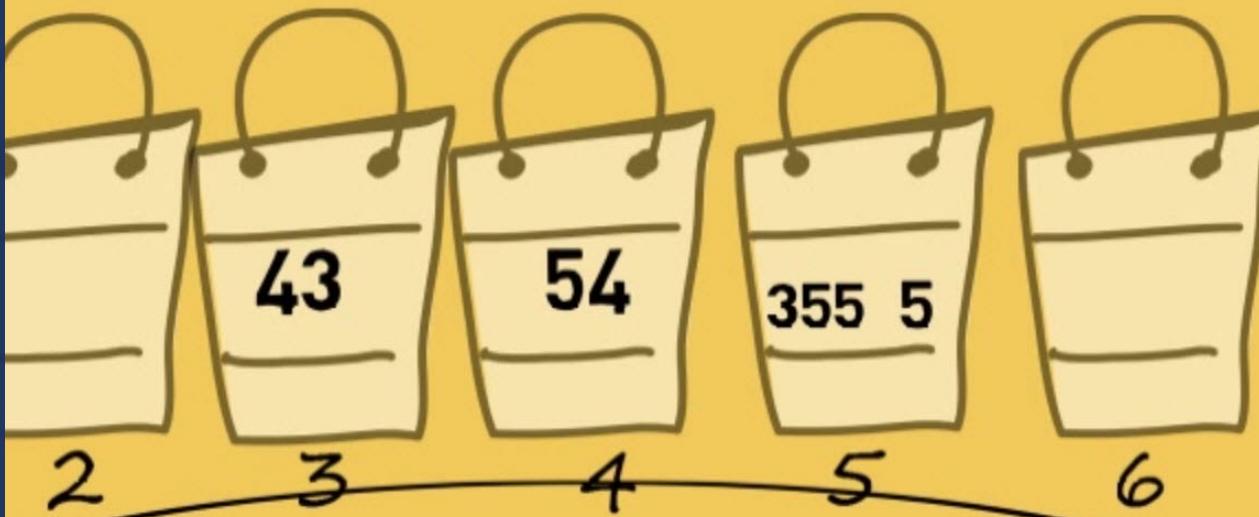
INPUT LIST TO BE SORTED

100 57
189

RADIX SORT

355 102 43 10

RADIX SORT



100 10 102 43 54 355 5 287

SORTED AS PER ONES PLACE VALUE

```
FUNC radix_sort(intList): DEV vacio
    max_digits ← length(string(max(intList)))
    exp ← 1
    digit ← 0
    base ← 10
    MIENTRAS digit < max_digits HACER
        bucket_sort(intList, exp, base)
        digit ← digit + 1
        exp ← exp * 10
    FIN MIENTRAS
FIN FUNC
```

```

FUNC bucket_sort(arr, exp, base): DEV vacio
    n <- length(arr)
    output <- array (length(n))
    count <- array (length(base))
    i <- 0: ENTERO
    MIENTRAS i < n HACER
        index <- (arr[i]//exp)
        count[(index)%base] <- count[(index)%base] + 1
        i <- i + 1
    FIN MIENTRAS
    i <- 1
    MIENTRAS i < base HACER
        count[i] <- count[i] + count[i-1]
        i <- i + 1
    FIN MIENTRAS
    i <- n-1
    MIENTRAS i>=0 HACER
        index <- (arr[i]//exp)
        output[count[(index)%base] - 1] <- arr[i]
        count[(index)%base] <- count[(index)%base] - 1
        i <- i - 1
    FIN MIENTRAS
    i <- 0
    MIENTRAS i < len(arr) HACER
        arr[i] <- output[i]
        i <- i + 1
    FIN MIENTRAS
FIN FUNC

```

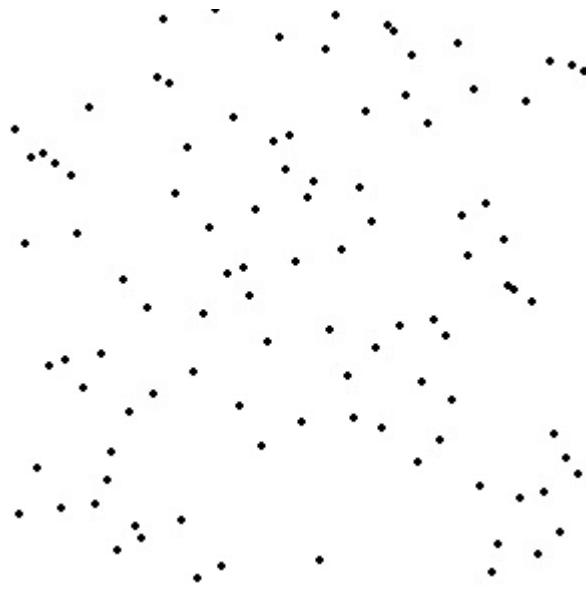


ALGORITMOS DE ORDENAMIENTO. PARTE 3.

PRÁCTICA 3

ALGORITMOS DE ORDENAMIENTO. PARTE 3.

- Implementar los algoritmos de Counting sort y Radix sort en Python para ordenar un nodo.
- Obtener los polinomios del mejor, el peor y el caso promedio de complejidad de cada algoritmo (Counting sort y Radix sort).
- Graficar en Python el comportamiento de los algoritmos implementados para diferentes instancias de tiempo (listas de 1 a 1000 elementos) para el mejor (lista ordenada), el peor (lista ordenada en forma inversa) y el caso promedio (lista aleatoria) de complejidad.



SELECTION SORT

By: Marco Polo
<https://commons.wikimedia.org/w/index.php?curid=6503932>

SELECTION SORT

El método de ordenamiento por selección consiste en buscar el menor elemento del conjunto y colocarlo en la primera posición. Luego se busca el siguiente elemento más pequeño y se coloca en la segunda posición. El proceso continúa hasta que todos los elementos del arreglo hayan sido ordenados.



El método se basa en los siguientes pasos:

- Seleccionar el menor elemento del conjunto.
- Intercambiar dicho elemento con el primero.
- Repetir los pasos anteriores para los $n-1$ elementos.

Ordenar el siguiente conjunto de forma ascendente utilizando el método selection sort.

A	15	67	8	16	44	27	12	35
---	----	----	---	----	----	----	----	----

Primer recorrido: se realiza la asignación menor $\leftarrow A[1]$.



menor < A[2]	$15 < 67$	Sí, no se realiza movimiento.
menor < A[3]	$15 < 8$	No, entonces menor $\leftarrow A[3]$
menor < A[4]	$8 < 16$	Sí, no se realiza movimiento.
menor < A[5]	$8 < 44$	Sí, no se realiza movimiento.
menor < A[6]	$8 < 27$	Sí, no se realiza movimiento.
menor < A[7]	$8 < 12$	Sí, no se realiza movimiento.
menor < A[8]	$8 < 35$	Sí, no se realiza movimiento.



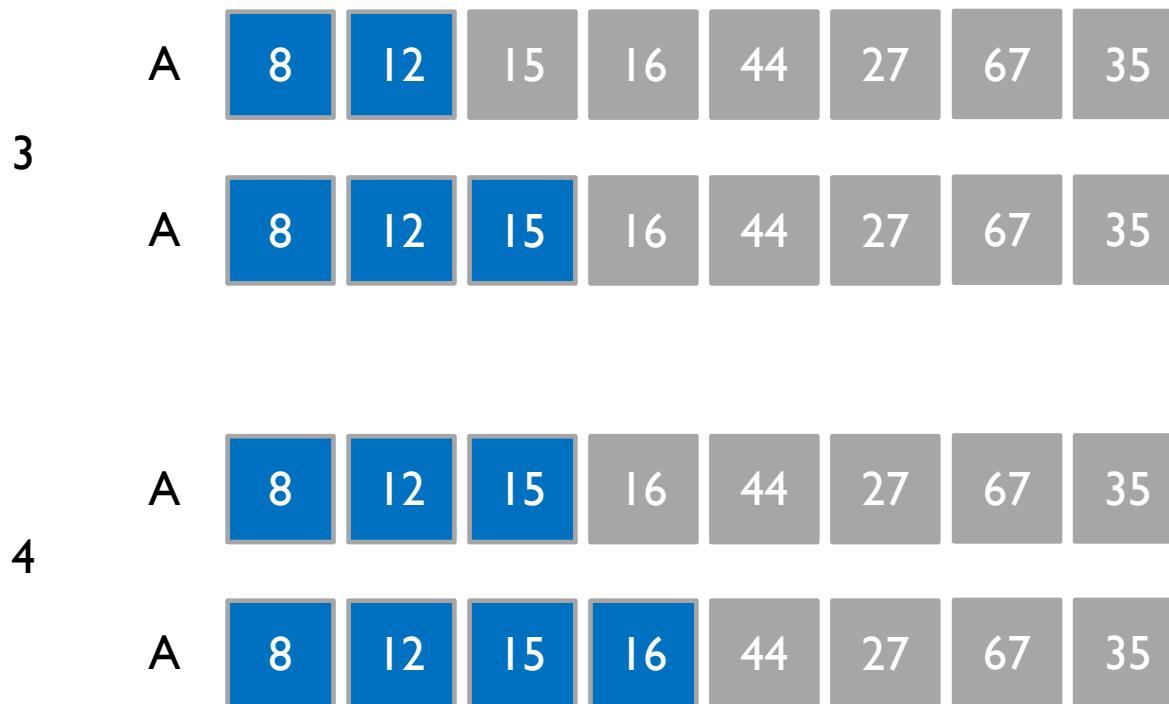
Segundo recorrido: se realiza la asignación menor $\leftarrow A[2]$.



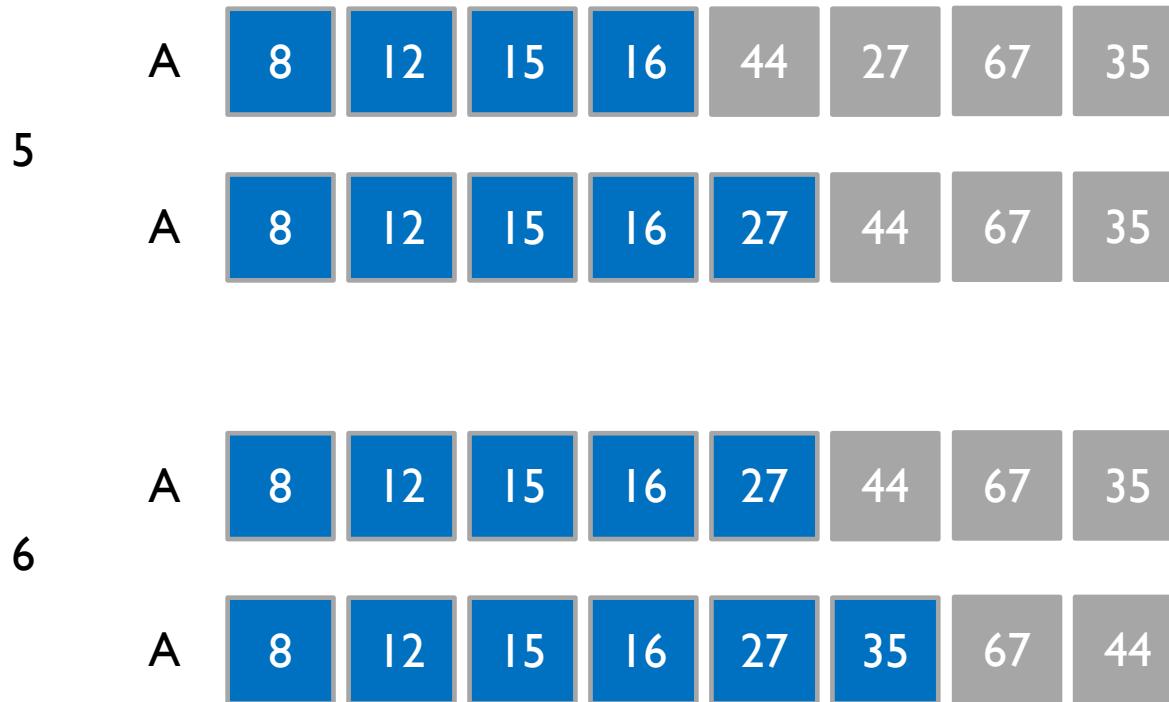
menor < A[3]	$67 < 15$	No, entonces menor $\leftarrow A[3]$
menor < A[4]	$15 < 16$	Sí, no se realiza movimiento.
menor < A[5]	$15 < 44$	Sí, no se realiza movimiento.
menor < A[6]	$15 < 27$	Sí, no se realiza movimiento.
menor < A[7]	$15 < 12$	No, entonces menor $\leftarrow A[7]$
menor < A[8]	$12 < 35$	Sí, no se realiza movimiento.



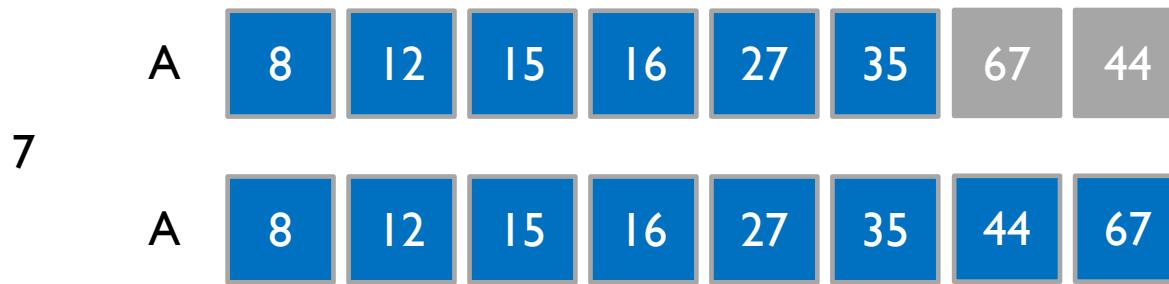
El recorrido continúa $n - 1$ veces.



El recorrido continúa $n - 1$ veces.



El recorrido continúa $n - 1$ veces.



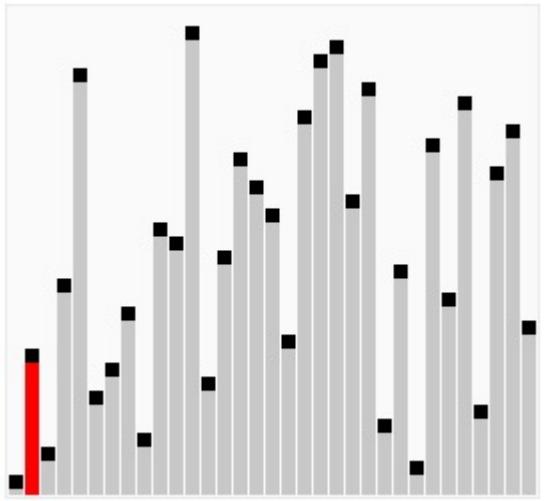
SELECTION SORT

203



El algoritmo de ordenamiento utilizando el método de selección es el siguiente:

```
FUNC selection_sort (n:Entero,A[]:Entero): DEV vacío
    cont ← 1, menor, aux: ENTERO
    MIENTRAS cont < n HACER
        menor ← A[cont] Y aux ← cont
        cont2 ← cont: Entero
        MIENTRAS cont2+1 < N HACER
            SI menor > A[cont2] HACER
                menor ← A[cont2]
                aux ← cont2
            FIN_SI
        FIN_MIENTRAS
        A[cont2] ← A[cont]
        A[cont] ← menor
    FIN_MIENTRAS
FIN_FUNC
```



SHAKER SORT

By Simpsons contributor - Own work, CC BY-SA 3.0
<https://commons.wikimedia.org/w/index.php?curid=14828123>

SHAKER SORT

El método shaker sort (método de la sacudida), es una optimización del método de intercambio directo. La idea de este método es combinar las dos maneras en que se puede realizar el método de la burbuja (intercambio directo).



Las iteraciones siguientes se realizan sobre el intervalo $[left, right]$. El algoritmo termina cuando en alguna iteración no se producen intercambios o bien cuando el elemento extremo izquierdo del conjunto ($left$) es mayor al contenido del elemento extremo derecho del conjunto ($right$).

Dentro del método de la sacudida se pueden identificar dos etapas distintas en cada iteración:

- La primera etapa consiste en recorrer los elementos de derecha a izquierda, moviendo los elementos más pequeños hacia la parte izquierda del conjunto. Se debe almacenar la posición del último elemento intercambiado (*left*).
- La segunda etapa consiste en recorrer los elementos de izquierda a derecha, moviendo los elementos más grandes hacia la parte derecha del conjunto. Se debe almacenar la posición del último elemento intercambiado (*right*).

Ordenar el siguiente conjunto de forma ascendente utilizando el método shaker sort.

A	15	67	8	16	44	27	12	35
---	----	----	---	----	----	----	----	----

Se realiza el primer recorrido de derecha a izquierda.



A[7]>A[8]	12 > 35	No hay intercambio.
A[6]>A[7]	27 > 12	Sí hay intercambio.
A[5]>A[6]	44 > 12	Sí hay intercambio.
A[4]>A[5]	16 > 12	Sí hay intercambio.
A[3]>A[4]	8 > 12	No hay intercambio.
A[2]>A[3]	67 > 8	Sí hay intercambio.
A[1]>A[2]	15 > 8	Sí hay intercambio.



left = 2

Se realiza el primer recorrido de izquierda a derecha.



$A[2] > A[3]$	$15 > 67$	No hay intercambio
$A[3] > A[4]$	$67 > 12$	Sí hay intercambio
$A[4] > A[5]$	$67 > 16$	Sí hay intercambio
$A[5] > A[6]$	$67 > 44$	Sí hay intercambio
$A[6] > A[7]$	$67 > 27$	Sí hay intercambio
$A[7] > A[8]$	$67 > 35$	Sí hay intercambio



right = 7

Se realiza el segundo recorrido de derecha a izquierda.



$A[6] > A[7]$	$27 > 35$	No hay intercambio
$A[5] > A[6]$	$44 > 27$	Sí hay intercambio
$A[4] > A[5]$	$16 > 27$	No hay intercambio
$A[3] > A[4]$	$12 > 16$	No hay intercambio
$A[2] > A[3]$	$15 > 12$	Sí hay intercambio



left = 3

Se realiza el segundo recorrido de izquierda a derecha.



$A[3] > A[4]$	$15 > 16$	No hay intercambio
$A[4] > A[5]$	$16 > 27$	No hay intercambio
$A[5] > A[6]$	$27 > 44$	No hay intercambio
$A[6] > A[7]$	$44 > 35$	Sí hay intercambio



right = 6

Se realiza el tercer recorrido de derecha a izquierda.

A

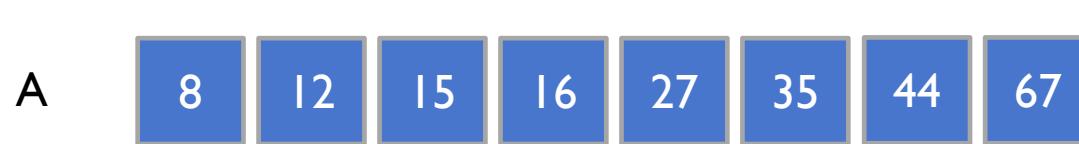


$$A[6] > A[7] \quad 27 > 35 \quad \text{No hay intercambio}$$

$$A[5] > A[6] \quad 16 > 27 \quad \text{No hay intercambio}$$

$$A[4] > A[5] \quad 15 > 16 \quad \text{No hay intercambio}$$

Debido a que en el tercer recorrido no se realizó intercambio alguno, la ejecución del algoritmo termina, por tanto, el conjunto está ordenado.



SHAKER SORT

216



El algoritmo de ordenamiento utilizando el método de la sacudida es el siguiente:

```
FUNC shaker_sort(n: Entero, A[]: Entero): DEV vacío
    izq ← 2, der ← n, pos ← n, aux, cont: ENTERO
    MIENTRAS(izq>der)HACER**Iniciaciclogeneral
        cont ← der
        **Recorrido de derecha a izquierda
        MIENTRAS cont > izq HACER
            SI A[cont - 1] > A[cont] ENTONCES
                aux ← A[cont-1]
                A[cont - 1] ← A[cont]
                A[cont] ← aux
                pos ← cont
            FIN_SI
        FIN_MIENTRAS
    izq ← pos - 1
```

El algoritmo de ordenamiento utilizando el método de la sacudida es el siguiente:

```
**Recorrido de izquierda a derecha
cont ← izq
MIENTRAS cont < der HACER
    SI A[cont - 1] > A[cont] ENTONCES
        aux ← A[cont - 1]
        A[cont - 1] ← A[cont]
        A[cont] ← aux
        pos ← cont
    FIN_SI
FIN_MIENTRAS
der ← pos - 1
FIN_MIENTRAS **Termina ciclo
FIN_FUNC
```



INSERTION SORT

By crashmatrix (talk | contribs)
<https://commons.wikimedia.org/w/index.php?curid=24614516>

INSERTION SORT

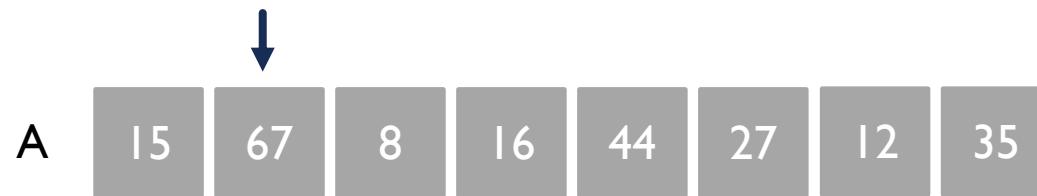
El método consiste en insertar un elemento del conjunto en la parte izquierda del mismo, el cual ya se encuentra previamente ordenado. El proceso se repite a partir del segundo elemento hasta el enésimo.

Este método es parecido al que utilizan los jugadores de cartas cuando acomodan (ordenan) sus jugadas, de ahí que también se le conoce como el método de la baraja.

Ordenar el siguiente conjunto de forma ascendente utilizando el método insertion sort (método de la baraja).

A	15	67	8	16	44	27	12	35
---	----	----	---	----	----	----	----	----

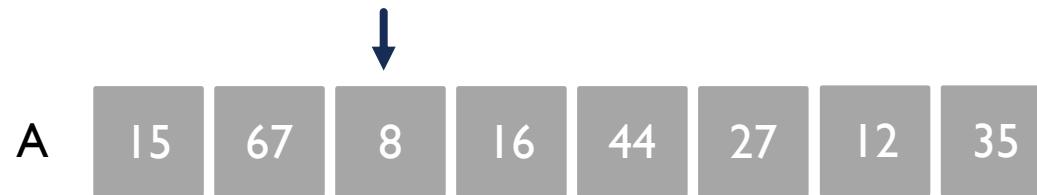
Primer recorrido



$A[2] < A[1]$ $67 < 15$ No hay intercambio



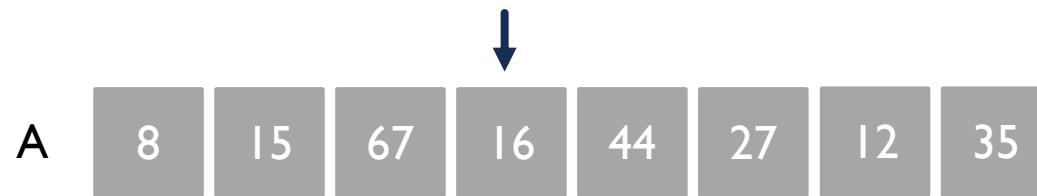
Segundo recorrido



$A[3] < A[2]$ $8 < 67$ Sí hay intercambio
 $A[2] < A[1]$ $8 < 15$ Sí hay intercambio



Tercer recorrido

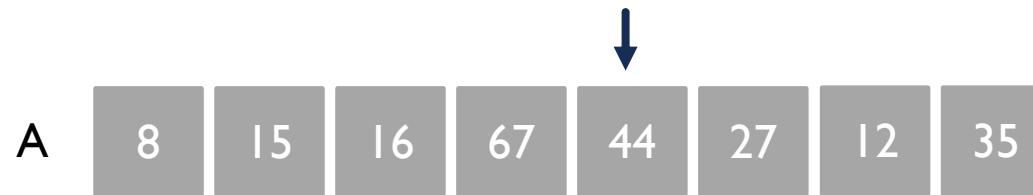


$$\begin{array}{ll} A[4] < A[3] & 16 < 67 \\ A[3] < A[2] & 16 < 15 \end{array}$$

Sí hay intercambio
No hay intercambio



Cuarto recorrido



$$A[5] < A[4] \quad 44 < 67$$

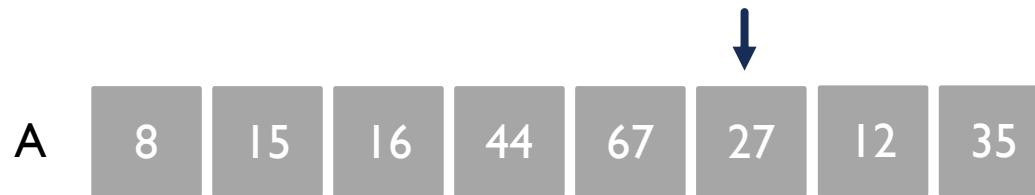
$$A[4] < A[3] \quad 44 < 16$$

Sí hay intercambio

No hay intercambio



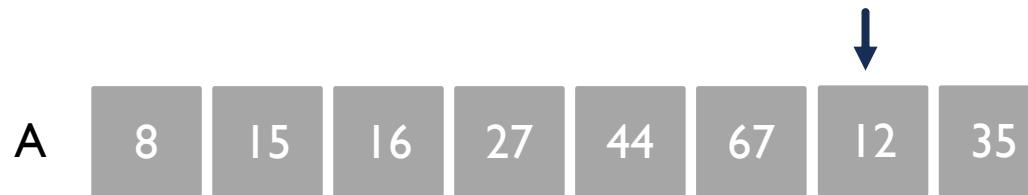
Quinto recorrido



$A[6] < A[5]$	$27 < 67$	Sí hay intercambio
$A[5] < A[4]$	$27 < 44$	Sí hay intercambio
$A[4] < A[3]$	$27 < 16$	No hay intercambio



Sexto recorrido



A[7] < A[6]	12 < 67	Sí hay intercambio
A[6] < A[5]	12 < 44	Sí hay intercambio
A[5] < A[4]	12 < 27	Sí hay intercambio
A[4] < A[3]	12 < 16	Sí hay intercambio
A[3] < A[2]	12 < 15	Sí hay intercambio
A[2] < A[1]	12 < 8	No hay intercambio



Séptimo recorrido



$$A[8] < A[7] \quad 35 < 67$$

Sí hay intercambio

$$A[7] < A[6] \quad 35 < 44$$

Sí hay intercambio

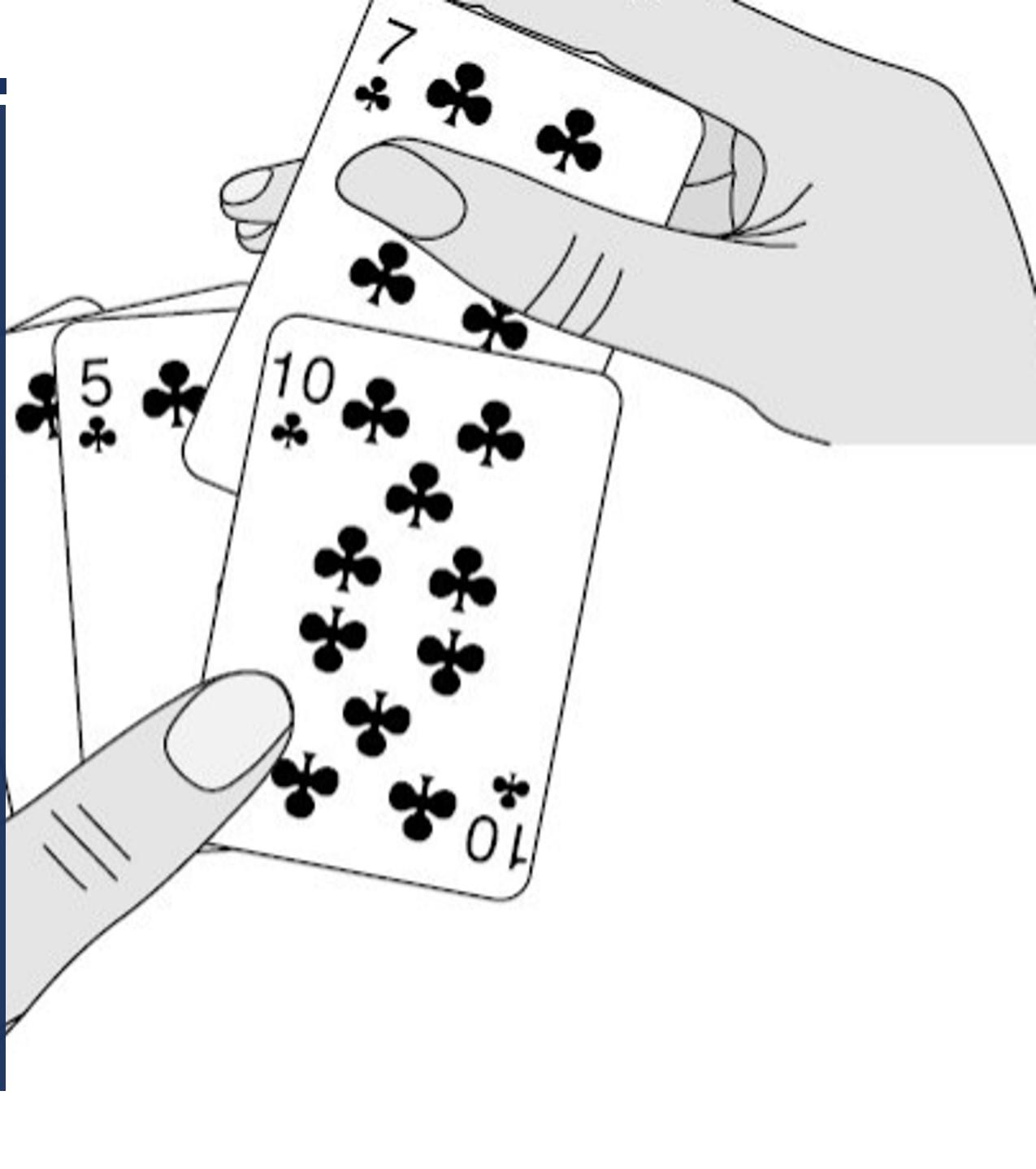
$$A[6] < A[5] \quad 35 < 27$$

No hay intercambio



INSERTION SORT

229





“Theory is when you know something, but it doesn’t work. Practice is when something works, but you don’t know why. Programmers combine theory and practice: Nothing works and they don’t know why.”

Unknown
@CodeWisdom

ORDENAMIENTOS EXTERNOS

Cuando el conjunto de datos que se desea clasificar es mayor a la capacidad de almacenamiento primaria y, por ende, los datos se encuentran almacenados en la memoria secundaria, los ordenamientos se realizan directamente sobre los archivos.

Suponiendo que se desea ordenar un conjunto de 5000 registros ($R_1, R_2, R_3, \dots, R_{5000}$), si las llaves de los registros solo pueden ser llevadas a la memoria principal en grupos de 1000, es posible dividir el conjunto en 5 subconjuntos, ordenarlos de forma independiente y, posteriormente, mezclarlos por intercalación.

Por tanto, los ordenamientos externos se basan en los ordenamientos internos vistos anteriormente.



El ordenamiento de archivos (externo) consta tres fases:

1. Fase de ordenamiento interno: en la cual se ordenan los registros mediante varias ejecuciones distribuidas en dos o más dispositivos de almacenamiento primario.
2. Fase de intercalación: en la cual se combinan los subarchivos ordenados en una sola ejecución.
3. Fase de salida: en la cual se copia el archivo ordenado en el medio de almacenamiento (secundario) final.



Todas las técnicas de ordenamiento de archivos operan esencialmente de la misma manera.

El conjunto de registros por ordenar se divide en varias listas, cada una de las cuales se ordenan mediante un método de ordenamiento interno. Cada lista ordenada se escribe como un archivo secuencial. Estos archivos ordenados se intercalan para formar un solo archivo ordenado.

Por tanto, a estas técnicas de ordenamiento se les conoce como técnicas de intercalación.



MÉTODO POR POLIFASE

ORDENAMIENTO EXTERNO

MÉTODO POR POLIFASE

Este método se apoya en los algoritmos de intercalación, por ello ha sido llamado ordenamiento de intercalación en polifase.

El método polifase es un tipo de intercalación desbalanceada, en la cual se utiliza un número constante de archivos auxiliares(T1, T2, T3 y T4), para almacenar la distribución de los elementos.



El ordenamiento polifase sigue 2 fases. En la primera fase se construyen los arreglos ordenados siguiendo los siguientes pasos:

1. Se leen las m llaves posibles.
2. Se ordenan las m llaves por un método interno.
3. Las m llaves se colocan en los archivos auxiliares T_2 y T_3 de manera intercalada (una lectura en T_2 y la otra en T_3).

En la segunda fase se intercalan los elementos de los archivos auxiliares hasta formar un solo archivo ordenado, es decir:

1. Se intercalan los primeros m elementos del primer archivo con los primeros m elementos del segundo archivo y se guardan en otro archivo auxiliar T_0 .
2. Se intercalan el siguiente bloque de m elementos del primer archivo con el siguiente bloque de m elementos del segundo archivo y se guardan en otro archivo auxiliar T_1 .
3. El proceso se repite hasta que los datos se agoten. El resultado de la intercalación se guarda en los primeros archivos auxiliares.

Se desea ordenar el siguiente conjunto de elementos que se encuentra en un archivo. El número máximo de llaves (m) que pueden ser llevadas a la memoria principal es 4.

T_0

10	42	50	80	20	15	19	70	78	69	55	8	14	30
----	----	----	----	----	----	----	----	----	----	----	---	----	----

Primera fase

 T_0  T_2  T_3  T_2  T_3 

Segunda fase

T_2	10	42	50	80	8	55	69	78						
T_3	15	19	20	70	14	30								
T_0	10	15	19	20	42	50	70	80						
T_1	8	14	30	55	69	78								

Segunda fase

T_0	10	15	19	20	42	50	70	80						
T_1	8	14	30	55	69	78								
T_2	8	10	14	15	19	20	30	42	50	55	69	70	78	80

El número total de recorridos (R) y comparaciones (C) que se realizan en el método de polifase, respectivamente, son:

$$R = \log(r) + i$$

$$C = n \log m + \sum_{i=1}^{\log(r)} n - \left\lfloor \frac{r}{2^i} \right\rfloor$$

donde:

n = número de elementos

m = número de elementos por arreglo

r = n / m



MÉTODO POR DISTRIBUCIÓN

ORDENAMIENTO EXTERNO

MÉTODO POR DISTRIBUCIÓN

El método de distribución para ordenamientos externos funciona de manera similar al método de ordenamiento interno.

Este método también es llamado ordenamiento digital. Una característica particular es que se opone a los conceptos de intercalación utilizados en otros ordenamientos externos.

Ordenar un conjunto de elementos numéricos base 4, es decir, expresiones que solo ocupan dígitos del 0 al 3.

Archivo
original

1023	0122	1131	3123	0012	0132	2013	1110
------	------	------	------	------	------	------	------

Se realizan 4 montones de elementos [0-3]. Cada elemento se lee de derecha a izquierda iniciando por el dígito más significativo (0).

Archivo
original

1023	0122	1131	3123	0012	0132	2013	1110
------	------	------	------	------	------	------	------

0	1	2	3
1110	1131	0122	1023
		0022	3123
		0132	2013

Después se extraen los montones de forma FIFO, es decir, de la columna uno se extraen todos los elementos, de la columna dos se extraen todos los elementos y así hasta la enésima columna, y se almacenan en un archivo auxiliar.

0	1	2	3
1110	1131	0122	1023
		0022	3123
		0132	2013

Archivo auxiliar



Se repite el proceso ahora con el archivo auxiliar y el siguiente dígito (1).

Archivo
original

1110	1131	0122	0012	0132	1023	3123	2013
------	------	------	------	------	------	------	------

0	1	2	3
	1110	0122	1131
	0012	1023	0132
	2013	3123	

Se vuelven a extraer los elementos columna por columna y en orden FIFO.

0	1	2	3
	1110	0122	1131
	0012	1023	0132
	2013	3123	

Archivo auxiliar



Se repite el proceso con el archivo auxiliar y el siguiente dígito (2).

Archivo
original

1110	0012	2013	0122	1023	3123	1131	0132
------	------	------	------	------	------	------	------

0	1	2	3
0012	1110		
2013	0122		
1023	3123		
	1131		
	0132		

Se vuelven a extraer los elementos columna por columna y en orden FIFO.

0	1	2	3
0012	1110		
2013	0122		
1023	3123		
	1131		
	0132		

Archivo auxiliar



Se repite el proceso con el archivo auxiliar y el siguiente dígito (3).

Archivo
original

0012	2013	1023	1110	0122	3123	1131	0132
------	------	------	------	------	------	------	------

0	1	2	3
0012	1023	2013	3123
0122	1110		
0132	1131		

Se vuelven a extraer los elementos columna por columna y en orden FIFO.

0	1	2	3
0012	1023	2013	3123
0122	1110		
0132	1131		

Archivo auxiliar



Cuando se recorren todos los dígitos de la llave, el archivo se encuentra ordenado.

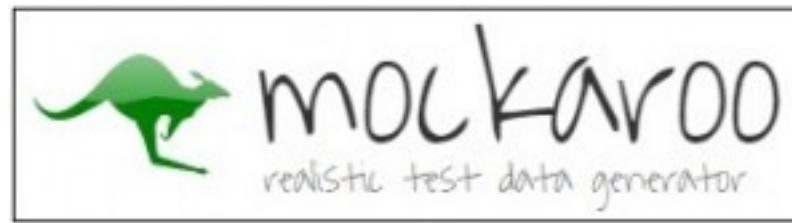
Archivo final

0012	0122	0132	1023	1110	1131	2013	3123
------	------	------	------	------	------	------	------

Cabe señalar que cada columna de la matriz representa un archivo auxiliar. Si se desea reducir el número de archivos se pueden transformar las llaves de los elementos a ordenar a otra base.

El tiempo total para el método por distribución está dado por el número de elementos del conjunto multiplicado por el número de dígitos de las llaves. Por tanto, el algoritmo ordena una lista en tiempo lineal, es decir, $O(n)$.

[HTTPS://MOCKAROO.COM](https://mockaroo.com)



“First do it, then do it right, then do it better.”

Addy Osmani

(Is an engineering manager at Google working on Chrome.)

I ALGORITMOS DE ORDENAMIENTO

Objetivo: Diseñar los métodos más importantes de algoritmos para efectuar ordenamientos en la computadora.

I.I Ordenamiento.

I.I.1 Bubble sort.

I.I.2 Heap sort.

I.I.3 Quick sort.

I.I.4 Countig sort.

I.I.5 Radix sort.

I.I.6 Merge sort.