UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 07: Ordenamientos II

ALUMNO:

Castañon Maldonado Carlos Emilio

PROFESORA

María de Luz Gasca Soto

AYUDANTES

Brenda Margarita Becerra Ruíz
Enrique Ehecatl Hernández Ferreiro (Link)

ASIGNATURA

Análisis de Algoritmos

1 El Problema de Selección consiste en encontrar el k-ésimo elemento más pequeño de un conjunto de n datos. Utilizar las estrategias usadas por el algoritmo Quick Sort, como el proceso Partition, para resolver el problema de Selección.
El algoritmo propuesto deberá tener desempeño computacional de $O(n)$, en el caso promedio.
Justifique con detalle sus respuestas.

Algoritmo :

(a) Seleccionar un elemento aleatorio del conjunto de datos como pivote.

(b) Realizar la operación de partición para dividir el conjunto de datos en dos subconjuntos: uno con elementos menores o iguales al pivote y otro con elementos mayores al pivote.

(c) Calcular la posición del pivote después de la partición, si la posición del pivote es igual a k, entonces hemos encontrado el k-ésimo elemento más pequeño y el algoritmo termina.

(d) Si la posición del pivote es mayor que k, repetir el proceso en el subconjunto de elementos menores que el pivote.

(e) Si la posición del pivote es menor que k, repetir el proceso en el subconjunto de elementos mayores que el pivote, pero actualizando k a $k-$(posición del pivote $+1$).

Justificación:

El algoritmo selecciona un pivote aleatorio en cada iteración, lo que garantiza que el algoritmo tenga un rendimiento promedio de O(n). Esto se debe a que, en promedio, el pivote elegido estará cerca del valor medio del conjunto de datos, dividiendo el conjunto aproximadamente a la mitad en cada iteración.

La operación de partición en el paso 2 se realiza en tiempo lineal $O(n)$ en el peor de los casos.

Después de cada partición, el algoritmo reduce efectivamente el tamaño del conjunto de datos en la mitad, ya que el subconjunto que contiene el k-ésimo elemento más pequeño se selecciona para futuras iteraciones.

El algoritmo repetirá la operación de partición en un subconjunto más pequeño hasta que se encuentre el k-ésimo elemento más pequeño, en cada iteración, el tamaño del conjunto de datos se reduce a la mitad, lo que resulta en un rendimiento de O(n) en el caso promedio.

2 Sea $QuickSort\_1$ la versión de Quick Sort que toma como pivote al elemento $A[(first + last)div2]$; y sea $QuickSort\_2$ la versión que toma como pivote al elemento que resulta ser la mediana de

$$A[first], A[(first + last)div2], A[last]$$

Dar un ejemplo de una lista de al menos 22 valores donde el desempeño computacional de $QuickSort\_2$ sea mejor que el de $QuickSort\_1$

Sea nuestro arreglo:
$A = \{99, 32, 89, 45, 73, 28, 91, 12, 64, 78, 53, 2, 1,$
$24, 42, 60, 86, 19, 5, 95, 8, 50, 56\}$

```
Quick Sort 1:

Arreglo original: [99, 32, 89, 45, 73, 28, 91, 12, 64, 78, 53, 2, 1, 24, 42, 60, 86, 19,
                   5, 95, 8, 50, 56]
Partición  1:  [1, 2] | 2 | [89, 45, 73, 28, 91, 12, 64, 78, 53, 32, 99, 24, 42, 60, 86,
                            19, 5, 95, 8, 50, 56]
Partición  2:  [1] | 1 | [2]
Partición  3:  [89, 45, 73, 28, 91, 12, 64, 78, 53, 32, 56, 24, 42, 60, 86, 19, 5, 95,
               8, 50] | 50 | [99]
Partición  4:  [8, 5, 19, 28, 24, 12, 32] | 32 | [78, 53, 64, 56, 91, 42, 60, 86, 73, 45,
                                                  95, 89, 50]
Partición  5:  [8, 5, 19, 12, 24] | 24 | [28, 32]
Partición  6:  [8, 5, 12] | 12 | [19, 24]
```

```
Partición  7:  [5] | 5 | [8, 12]
Partición  8:  [8] | 8 | [12]
Partición  9:  [19] | 19 | [24]
Partición 10:  [28] | 28 | [32]
Partición 11:  [50, 53, 45, 56, 60, 42] | 42 | [91, 86, 73, 64, 95, 89, 78]
Partición 12:  [42, 45] | 45 | [53, 56, 60, 50]
Partición 13:  [42] | 42 | [45]
Partición 14:  [53, 50] | 50 | [60, 56]
Partición 15:  [50] | 50 | [53]
Partición 16:  [56] | 56 | [60]
Partición 17:  [64] | 64 | [86, 73, 91, 95, 89, 78]
Partición 18:  [86, 73, 78, 89] | 89 | [95, 91]
Partición 19:  [73] | 73 | [86, 78, 89]
Partición 20:  [78] | 78 | [86, 89]
Partición 21:  [86] | 86 | [89]
Partición 22:  [91] | 91 | [95]
Arreglo ordenado: [1, 2, 5, 8, 12, 19, 24, 28, 32, 42, 45, 50, 53, 56, 60, 64, 73, 78,
                   86, 89, 91, 95, 99]
```

```
Quick Sort 2:

Arreglo original: [99, 32, 89, 45, 73, 28, 91, 12, 64, 78, 53, 2, 1, 24, 42, 60, 86, 19,
                   5, 95, 8, 50, 56]
Partición  1: [56, 32, 50, 45, 8, 28, 5, 12, 19, 42, 53, 2, 1, 24] | 24 | [78, 60, 86, 64,
                                                                      91, 95, 73, 89, 99]
Partición  2: [24, 1, 2, 19, 8, 12, 5] | 5 | [28, 45, 42, 53, 50, 32, 56]
Partición  3: [5, 1, 2, 12, 8] | 8 | [19, 24]
Partición  4: [2, 1] | 1 | [5, 12, 8]
Partición  5: [1] | 1 | [2]
Partición  6: [5, 8] | 8 | [12]
Partición  7: [5] | 5 | [8]
Partición  8: [19] | 19 | [24]
Partición  9: [28, 45, 42, 32, 50] | 50 | [53, 56]
Partición 10: [28, 32, 42] | 42 | [45, 50]
Partición 11: [28, 32] | 32 | [42]
Partición 12: [28] | 28 | [32]
Partición 13: [45] | 45 | [50]
Partición 14: [53] | 53 | [56]
Partición 15: [78, 60, 86, 64, 89, 73] | 73 | [95, 91, 99]
Partición 16: [73, 60, 64] | 64 | [86, 89, 78]
Partición 17: [64, 60] | 60 | [73]
Partición 18: [60] | 60 | [64]
Partición 19: [78] | 78 | [89, 86]
Partición 20: [86] | 86 | [89]
Partición 21: [91] | 91 | [95, 99]
Partición 22: [95] | 95 | [99]
Arreglo ordenado: [1, 2, 5, 8, 12, 19, 24, 28, 32, 42, 45, 50, 53, 56, 60, 64, 73, 78,
                   86, 89, 91, 95, 99]
```

Como podemos observar, el arreglo dado, favorece en desempeño computacional a Quick Sort 2 ya que el desempeño de este entra en el mejor de los casos de Quick Sort, en cambio, Quick Sort 1 entra en el peor de los casos desde sus primeras tres particiones.

3 Proporcione una secuencia L de enteros diferentes, de tres dígitos cada uno.
Considere $|L| \geq 30$
Aplique Bucket Sort a L de dos maneras distintas.

$L = \{764, 235, 129, 492, 631, 845, 376, 508, 347, 912, 586, 419, 753, 624, 187, 265, 978, 341, 562, 829, 154, 496, 723, 618, 295, 871, 430, 569, 324, 784\}$

a)

Para esta aplicación, tendremos diez cubetas para cada familia de centenares:

Elementos asignados en el bucket.

```
 _
|B|
---
|0| = []
---
---
|1| = [129 -> 187 -> 154]
---
---
|2| = [235 -> 265 -> 295]
---
---
|3| = [376 -> 347 -> 341 -> 324]
---
---
|4| = [492 -> 419 -> 496 -> 430]
---
---
|5| = [508 -> 586 -> 562 -> 569]
---
---
|6| = [631 -> 624 -> 618]
---
---
|7| = [764 -> 753 -> 723 -> 784]
---
---
|8| = [845 -> 829 -> 871]
---
---
|9| = [912 -> 978]
---
```

Elementos del bucket ordenados.

```
 _
|B|
---
|0| = []
---
---
|1| = [129 -> 154 -> 187]
---
---
|2| = [235 -> 265 -> 295]
---
---
|3| = [324 -> 341 -> 347 -> 376]
---
---
|4| = [419 -> 430 -> 492 -> 496]
---
---
|5| = [508 -> 562 -> 569 -> 586]
---
---
|6| = [618 -> 624 -> 631]
---
---
|7| = [723 -> 753 -> 764 -> 784]
---
---
|8| = [829 -> 845 -> 871]
---
---
|9| = [912 -> 978]
---
```

Arreglo ordenado:
$L' = \{129, 154, 187, 235, 265, 295, 324, 341, 347, 376, 419, 430, 492, 496, 508, 562, 569, 586, 618, 624, 631, 723, 753, 764, 784, 829, 845, 871, 912, 978'\}$

b)

Para esta aplicación, tendremos cinco cubetas:

$L = \{764, 235, 129, 492, 631, 845, 376, 508, 347, 912, 586, 419, 753, 624, 187, 265, 978, 341, 562, 829, 154, 496, 723, 618, 295, 871, 430, 569, 324, 784\}$

Elementos asignados en el bucket.

```
 _
|B|
---
|0| = [129 -> 187 -> 154]
---
---
|1| = [235 -> 376 -> 347 -> 265 ->
---    341 -> 295 -> 324]
---
|2| = [492 -> 508 -> 586 -> 419 ->
---    562 -> 496 -> 430 -> 569]
---
|3| = [764 -> 631 -> 753 -> 624 ->
---    723 -> 618 -> 784]
---
|4| = [845 -> 912 -> 978 -> 829 ->
---    871]
```

Elementos del bucket ordenados.

```
 _
|B|
---
|0| = [129 -> 154 -> 187]
---
---
|1| = [235 -> 265 -> 295 -> 324 ->
---    341 -> 347 -> 376]
---
|2| = [419 -> 430 -> 492 -> 496 ->
---    508 -> 562 -> 569 -> 586]
---
|3| = [618 -> 624 -> 631 -> 723 ->
---    753 -> 764 -> 784]
---
|4| = [829 -> 845 -> 871 -> 912 ->
---    978]
```

Arreglo ordenado:

$L' = \{129, 154, 187, 235, 265, 295, 324, 341, 347, 376, 419, 430, 492, 496, 508, 562, 569, 586, 618, 624, 631, 723, 753, 764, 784, 829, 845, 871, 912, 978'\}$

4 Proporcione una secuencia L de enteros diferentes, en hexadecimal, de cuatro cifras cada uno. Considere $|L| \geq 25$

a) Ordene la secuencia usando...
i) ... MSD-Radix-Sort;
ii) ... LSD-Radix-Sort.

El arreglo que estaremos utilizando para las ejecuciones de MSD-Radix-Sort y de LSD-Radix-Sort será el siguiente;

$$\begin{aligned} L = [&0xe677, 0xa3e6, 0x44c1, 0x6dec, 0x57ca, \\ &0xea5f, 0x9f8d, 0x8b23, 0xd590, 0x2e16, \\ &0x3c97, 0x7261, 0xf1d8, 0x5a42, 0x846e, \\ &0x2f43, 0xcba5, 0x65fd, 0x92e7, 0x4d8c, \\ &0x369f, 0xfeca, 0xae57, 0x81b9, 0xd241] \end{aligned}$$

Notese que en las siguientes ejecuciones tendremos un total de 16 *Buckets* por el formato de nuestros numeros Hexadecimales, si estos fueran Decimales, bastarían precisamente solo 10 *Buckets*.
También notemos que para un numero hexadecimal cualquiera como $e677$, este debe estar acompañado del sufijo $0x$, esto por norma y por que nos indica que esto $0xe677$ es un numero hexadecimal.

i) MSD-Radix-Sort;

$$\begin{aligned} L = [&0xe677, 0xa3e6, 0x44c1, 0x6dec, 0x57ca,\\ &0xea5f, 0x9f8d, 0x8b23, 0xd590, 0x2e16,\\ &0x3c97, 0x7261, 0xf1d8, 0x5a42, 0x846e,\\ &0x2f43, 0xcba5, 0x65fd, 0x92e7, 0x4d8c,\\ &0x369f, 0xfeca, 0xae57, 0x81b9, 0xd241] \end{aligned}$$

Iteración: 01

```
 _
|B|

|0| = []

|1| = []

|2| = ['0x2e16', '0x2f43']

|3| = ['0x3c97', '0x369f']

|4| = ['0x44c1', '0x4d8c']

|5| = ['0x57ca', '0x5a42']

|6| = ['0x6dec', '0x65fd']

|7| = ['0x7261']

|8| = ['0x8b23', '0x846e', '0x81b9']

|9| = ['0x9f8d', '0x92e7']

|10| = ['0xa3e6', '0xae57']

|11| = []

|12| = ['0xcba5']

|13| = ['0xd590', '0xd241']

|14| = ['0xe677', '0xea5f']

|15| = ['0xf1d8', '0xfeca']
```

BASE 10	BASE 16
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Para esta primera iteración, es bastante trivial el como hemos acomodado cada uno de nuestros numeros hex en las cubetas (buckets), como estamos trabajando con Radix Most Significant Digit, primero tomamos el primer numero de nuestro arreglo y nos fijamos en su dígito mas significativo, en este caso fue $0xe677$, nos damos cuenta que su MSD es e, en base a esto y revisando nuestra tabla de conversión decimal a hex, podemos saber que este numero va en la cubeta 14, este proceso se va a repetir para todos los numeros del arreglo respetando siempre el orden de aparición de los mismos.

```
Iteración: 02

 _
|B|

|0| = []
|1| = []
|2| = []
|3| = []
|4| = []
|5| = []
|6| = []
|7| = []
|8| = []
|9| = []
|10| = []
|11| = []
|12| = []
|13| = []
|14| = ['0x2e16']
|15| = ['0x2f43']
```

```
Iteración: 03

 _
|B|

|0| = []
|1| = []
|2| = []
|3| = []
|4| = []
|5| = []
|6| = ['0x369f']
|7| = []
|8| = []
|9| = []
|10| = []
|11| = []
|12| = ['0x3c97']
|13| = []
|14| = []
|15| = []
```

```
Iteración: 04

 _
|B|

|0| = []
|1| = []
|2| = []
|3| = []
|4| = ['0x44c1']
|5| = []
|6| = []
|7| = []
|8| = []
|9| = []
|10| = []
|11| = []
|12| = []
|13| = ['0x4d8c']
|14| = []
|15| = []
```

```
Iteración: 05

 _
|B|

|0| = []
|1| = []
|2| = []
|3| = []
|4| = []
|5| = []
|6| = []
|7| = ['0x57ca']
|8| = []
|9| = []
|10| = ['0x5a42']
|11| = []
|12| = []
|13| = []
|14| = []
|15| = []
```

```
Iteración: 06

 _
|B|

|0| = []
|1| = []
|2| = []
|3| = []
|4| = []
|5| = ['0x65fd']
|6| = []
|7| = []
|8| = []
|9| = []
|10| = []
|11| = []
|12| = []
|13| = ['0x6dec']
|14| = []
|15| = []
```

```
Iteración: 07

 _
|B|

|0| = []
|1| = ['0x81b9']
|2| = []
|3| = []
|4| = ['0x846e']
|5| = []
|6| = []
|7| = []
|8| = []
|9| = []
|10| = []
|11| = ['0x8b23']
|12| = []
|13| = []
|14| = []
|15| = []
```

Notemos como es que solo hemos iterado sobre aquellas cubetas que tengan 2 o mas elementos (dejando afuera las que no tienen elementos o las que solo tenían un elemento) esto con el fin de ordenar los elementos con su siguiente MSD, notemos como es que en la primera iteración teníamos por ejemplo a la cubeta $|3|$ con los elementos $[0x3c97, 0x369f]$, el siguiente MSD de $3c97$ es c y el siguiente MSD de $369f$ es 6, con esta información procedemos a poner nuestros elementos en la respectiva cubeta a la que le corresponda este MSD, notese que en todas estas iteraciones y las siguientes vamos a repetir este proceso y en caso de que alguna de estas sub-cubetas de las cubetas originales resultara tener otra vez mas de 1 elemento repetiríamos este proceso.
También notemos como es que a partir de la iteración 2 estamos operando en sub cubetas y no en las cubetas originales.

```
Iteración: 08

 _
|B|

|0| = []

|1| = []

|2| = ['0x92e7']

|3| = []

|4| = []

|5| = []

|6| = []

|7| = []

|8| = []

|9| = []

|10| = []

|11| = []

|12| = []

|13| = []

|14| = []

|15| = ['0x9f8d']
```

```
Iteración: 09

 _
|B|

|0| = []

|1| = []

|2| = []

|3| = ['0xa3e6']

|4| = []

|5| = []

|6| = []

|7| = []

|8| = []

|9| = []

|10| = []

|11| = []

|12| = []

|13| = []

|14| = ['0xae57']

|15| = []
```

```
Iteración: 10

 _
|B|

|0| = []

|1| = []

|2| = ['0xd241']

|3| = []

|4| = []

|5| = ['0xd590']

|6| = []

|7| = []

|8| = []

|9| = []

|10| = []

|11| = []

|12| = []

|13| = []

|14| = []

|15| = []
```

```
Iteración: 11

 _
|B|

|0| = []

|1| = []

|2| = []

|3| = []

|4| = []

|5| = []

|6| = ['0xe677']

|7| = []

|8| = []

|9| = []

|10| = ['0xea5f']

|11| = []

|12| = []

|13| = []

|14| = []

|15| = []
```

```
Iteración: 12

 _
|B|

|0| = []

|1| = ['0xf1d8']

|2| = []

|3| = []

|4| = []

|5| = []

|6| = []

|7| = []

|8| = []

|9| = []

|10| = []

|11| = []

|12| = []

|13| = []

|14| = ['0xfeca']

|15| = []
```

Como hemos podido apreciar, en nuestras iteraciones hemos cambiado el orden de nuestras sub-cubetas y por ende el de nuestras cubetas originales, quedando estas de la siguiente manera:

```
 _
|B|

|0| = []

|1| = []

|2| = ['0x2e16', '0x2f43']

|3| = ['0x369f', '0x3c97']

|4| = ['0x44c1', '0x4d8c']

|5| = ['0x57ca', '0x5a42']

|6| = ['0x65fd', '0x6dec']

|7| = ['0x7261']

|8| = ['0x81b9', '0x846e', '0x8b23']

|9| = ['0x92e7', '0x9f8d']

|10| = ['0xa3e6', '0xae57']

|11| = []

|12| = ['0xcba5']

|13| = ['0xd241', '0xd590']

|14| = ['0xe677', '0xea5f']

|15| = ['0xf1d8', '0xfeca']
```

Como paso final, ya solo queda concatenar cada cubeta para obtener el Arreglo Ordenado:

Arreglo Ordenado:

$$\begin{aligned} L = [&0x2e16, 0x2f43, 0x369f, 0x3c97, 0x44c1, \\ &0x4d8c, 0x57ca, 0x5a42, 0x65fd, 0x6dec, \\ &0x7261, 0x81b9, 0x846e, 0x8b23, 0x92e7, \\ &0x9f8d, 0xa3e6, 0xae57, 0xcba5, 0xd241, \\ &0xd590, 0xe677, 0xea5f, 0xf1d8, 0xfeca] \end{aligned}$$

ii) LSD-Radix-Sort;

$$\begin{aligned} L = [&0xe677, 0xa3e6, 0x44c1, 0x6dec, 0x57ca,\\ &0xea5f, 0x9f8d, 0x8b23, 0xd590, 0x2e16,\\ &0x3c97, 0x7261, 0xf1d8, 0x5a42, 0x846e,\\ &0x2f43, 0xcba5, 0x65fd, 0x92e7, 0x4d8c,\\ &0x369f, 0xfeca, 0xae57, 0x81b9, 0xd241] \end{aligned}$$

Iteración: 01

```
 _
|B|

|0| = ['0xd590']

|1| = ['0x44c1', '0x7261', '0xd241']

|2| = ['0x5a42']

|3| = ['0x8b23', '0x2f43']

|4| = []

|5| = ['0xcba5']

|6| = ['0xa3e6', '0x2e16']

|7| = ['0xe677', '0x3c97', '0x92e7',
       '0xae57']

|8| = ['0xf1d8']

|9| = ['0x81b9']

|10| = ['0x57ca', '0xfeca']

|11| = []

|12| = ['0x6dec', '0x4d8c']

|13| = ['0x9f8d', '0x65fd']

|14| = ['0x846e']

|15| = ['0xea5f', '0x369f']
```

Iteración: 02

```
 _
|B|

|0| = []

|1| = ['0x2e16']

|2| = ['0x8b23']

|3| = []

|4| = ['0xd241', '0x5a42', '0x2f43']

|5| = ['0xae57', '0xea5f']

|6| = ['0x7261', '0x846e']

|7| = ['0xe677']


|8| = ['0x4d8c', '0x9f8d']

|9| = ['0xd590', '0x3c97', '0x369f']

|10| = ['0xcba5']

|11| = ['0x81b9']

|12| = ['0x44c1', '0x57ca', '0xfeca']

|13| = ['0xf1d8']

|14| = ['0xa3e6', '0x92e7', '0x6dec']

|15| = ['0x65fd']
```

Tal y como su nombre lo indica, Radix Least Significant Digit va a tomar el arreglo original y lo va a ordenar primeramente por sus dígitos menos significativos (y su orden de aparición), teniendo ahora que al por ejemplo encontrarse con $0xa3e6$, puede apreciarse que su LSD es 6 (y por ende se coloca en la cubeta 6), este proceso se repite para todos los numeros para la primera iteración, para la segunda iteración repetimos el proceso para el siguiente LSD de todos los numeros, este proceso se repite hasta que hayamos ordenado el ultimo LSD (que vendría siendo el MSD) del numero mas grande en la ultima iteración.

Iteración: 03

```
 _
|B|

|0| = []

|1| = ['0x81b9', '0xf1d8']

|2| = ['0xd241', '0x7261', '0x92e7']

|3| = ['0xa3e6']

|4| = ['0x846e', '0x44c1']

|5| = ['0xd590', '0x65fd']

|6| = ['0xe677', '0x369f']

|7| = ['0x57ca']

|8| = []

|9| = []

|10| = ['0x5a42', '0xea5f']

|11| = ['0x8b23', '0xcba5']

|12| = ['0x3c97']

|13| = ['0x4d8c', '0x6dec']

|14| = ['0x2e16', '0xae57', '0xfeca']

|15| = ['0x2f43', '0x9f8d']
```

Iteración: 04

```
 _
|B|

|0| = []

|1| = []

|2| = ['0x2e16', '0x2f43']

|3| = ['0x369f', '0x3c97']

|4| = ['0x44c1', '0x4d8c']

|5| = ['0x57ca', '0x5a42']

|6| = ['0x65fd', '0x6dec']

|7| = ['0x7261']

|8| = ['0x81b9', '0x846e', '0x8b23']

|9| = ['0x92e7', '0x9f8d']

|10| = ['0xa3e6', '0xae57']

|11| = []

|12| = ['0xcba5']

|13| = ['0xd241', '0xd590']

|14| = ['0xe677', '0xea5f']

|15| = ['0xf1d8', '0xfeca']
```

Arreglo Ordenado:

$$\begin{aligned} L = [&0x2e16, 0x2f43, 0x369f, 0x3c97, 0x44c1,\\ &0x4d8c, 0x57ca, 0x5a42, 0x65fd, 0x6dec,\\ &0x7261, 0x81b9, 0x846e, 0x8b23, 0x92e7,\\ &0x9f8d, 0xa3e6, 0xae57, 0xcba5, 0xd241,\\ &0xd590, 0xe677, 0xea5f, 0xf1d8, 0xfeca] \end{aligned}$$

Notemos como es que LSD no necesita de mas cubetas y este siempre opera con las cubetas originales, además de que aunque nosotros no lo veamos, con cada iteración se genera un arreglo nuevo con el cual va a operar la siguiente iteración con el siguiente LSD.

5 Opcional Sea L una lista de n números enteros diferentes.
Suponga que los elementos x de L están en el intervalo $[1, 500]$.
Diseñe un algoritmo de orden lineal que ordene los elementos de L.

Algoritmo:

(a) Creamos un arreglo de conteo con un tamaño igual al rango de valores que se esperan en la lista.

(b) Recorremos la lista original y contamos la frecuencia de cada elemento, para cada elemento x, incrementar el contador en la posición correspondiente en el arreglo de conteo.

(c) Reconstruir la lista ordenada, comenzando desde el valor más bajo del rango (en este caso, 1) y recorremos el arreglo de conteo, por cada elemento en el arreglo de conteo, agregamos i a la lista ordenada $count[i]$ veces, donde i es el valor del elemento en la posición i del arreglo de conteo.

(d) Al final del proceso, tendremos una lista ordenada.

```
def algoritmo(arr):
    -- Crear un arreglo de conteo con 500 elementos (0 a 500)
    count <- [0] * 501
    n <- len(arr)

    -- Contar la frecuencia de cada elemento en arr
    for num in arr:
        count[num] += 1

    -- Reconstruir la lista ordenada
    arrOrd <- []
    for i in range(1, 501):
        while count[i] > 0:
            arrOrd.append(i)
            count[i] -= 1

    return arrOrd
```