ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA

UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

ESTRUCTURA DE DATOS

Ing. Mayra Alvarez, MSc.
mialvarez2@espe.edu.ec

Lección 2.1.2 : Algoritmos de Ordenación Interna Indirectos o Avanzados

Definición

- En lugar de reorganizar los elementos en la memoria (directos), estos algoritmos trabajan con una lista de índices que representan la posición original de los elementos en el conjunto.
- En este enfoque, se crea un arreglo adicional que contiene los índices de los elementos originales. Luego, el algoritmo de ordenación trabaja sobre este arreglo de índices, reorganizándolos según el criterio de ordenación deseado.
- Finalmente, se utiliza el arreglo de índices ordenado para acceder a los elementos originales en el orden especificado.

Lección 2.1.3 : Quicksort

Método Ordenación Rápida (Quicksort)

- El algoritmo se basa en la división de la lista/arreglo en particiones a ordenar, aplicando la técnica "***divide y vencerás***".
- Divide le problema original en subproblemas, resuelve los subproblemas de forma ***recursiva***, combina las soluciones de los subproblemas para llegar a la solución final.
- La combinación de las soluciones se da de forma automática.
- El método es, posiblemente, el más eficiente de los algoritmos conocidos de ordenación.
- Complejidad del algoritmo O(n log(n)), en el mejor de los casos.

Método Quicksort

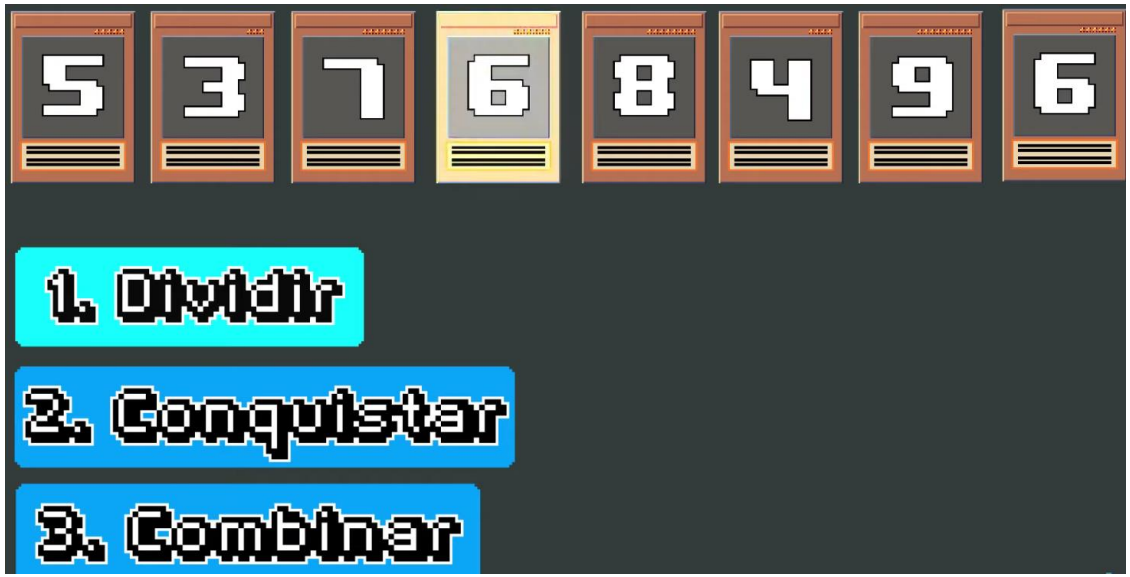
1. Divide los **n** elementos de la lista en dos partes separadas por un elemento: una partición izquierda, un elemento central "***pivote***", y una partición derecha.
2. El pivote se debe elegir de modo que se divida la lista por la mitad, de acuerdo al tamaño relativo de las claves (1er elemento, ultimo, centro, aleatoreo).
3. La partición se hace de tal forma que todos los elementos de la 1ra sublista (izquierda) son menores que todos los elementos de la 2da sublista (derecha).
4. Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo.

Método Quicksort

En el canal de YouTube de Chio Code encontramos la descripción del método Quick Sort
https://www.youtube.com/watch?v=WprjBK0p6rw

Método Quicksort

Elegir un elemento de la lista de elementos a ordenar, al que llamaremos **PIVOTE**. Además, será necesario 2 punteros que van a recorrer las posiciones del arreglo.



Método Quicksort

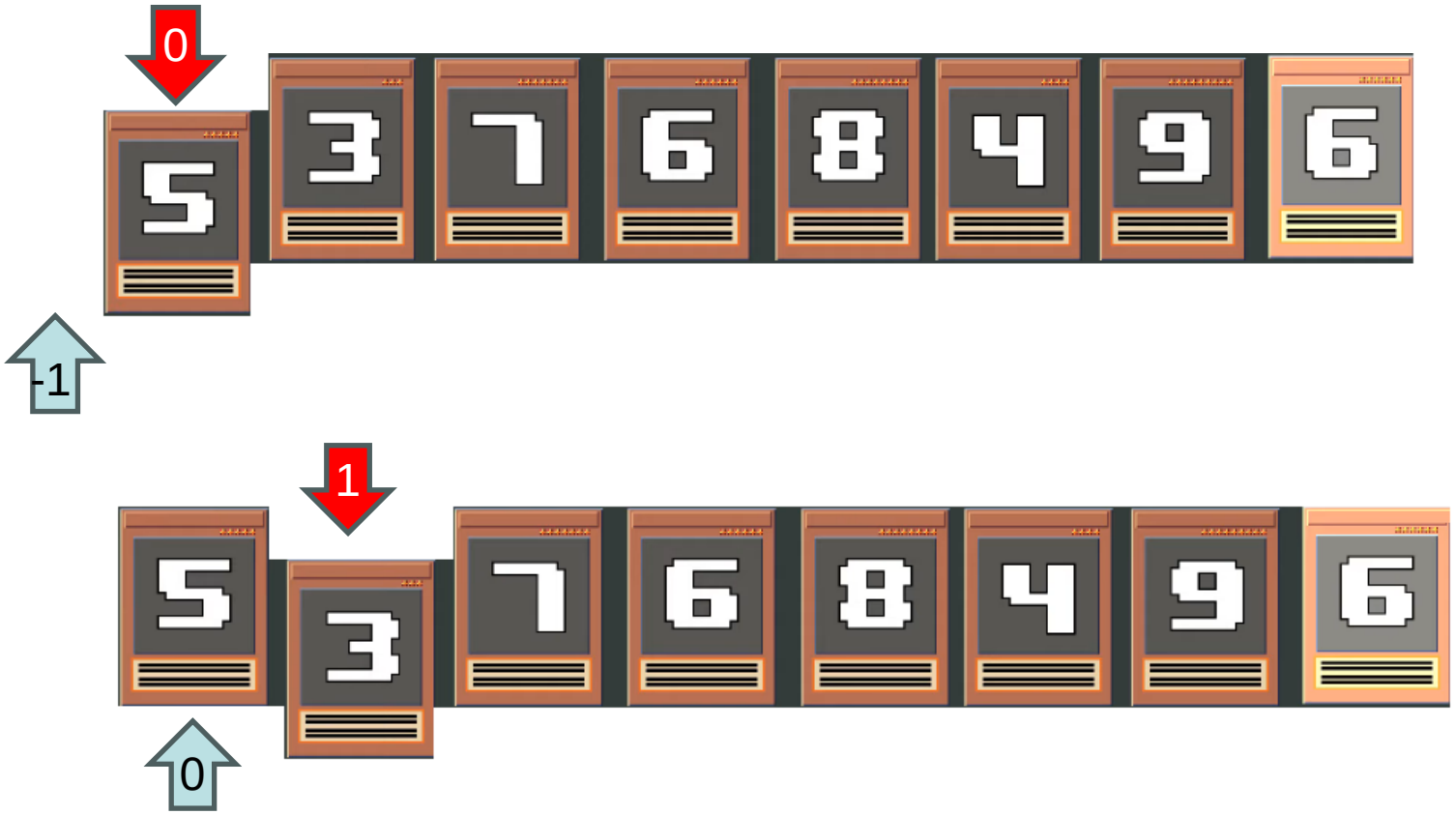
PIVOTE como **ultimo** elemento.

```
if arr[i] < pivote:
  temp = arr[peque+1]
  arr[peque+1] = arr[i]
  arr[i] = temp
  peque++
```

(Recorre todo el arreglo)

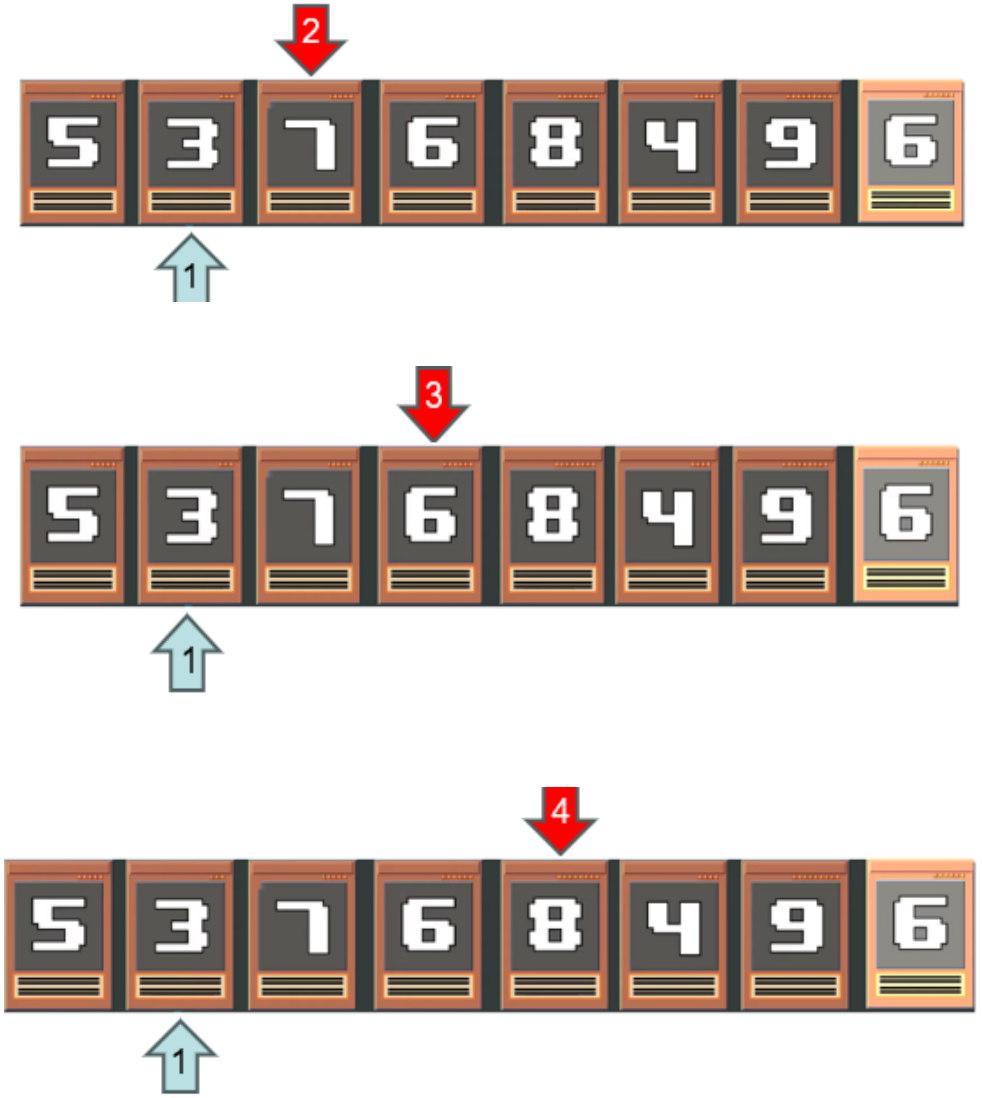
-1 (Último elemento mas pequeño que el PIVOTE)

Método Quicksort



Algoritmos de Ordenación Interna

Método Quicksort



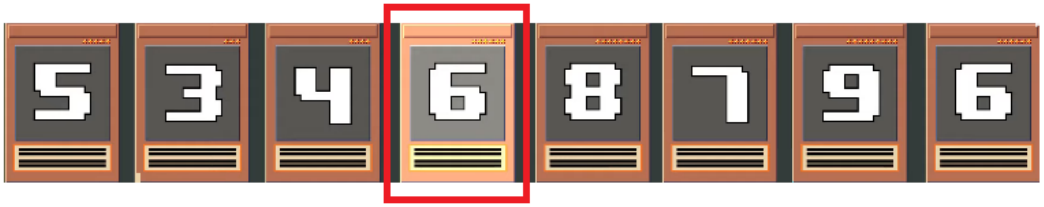
Método Quicksort

El elemento 4 es menor que PIVOTE.

Incrementamos el ultimo elemento mas pequeño (index).

Intercambiamos valores.

Método Quicksort

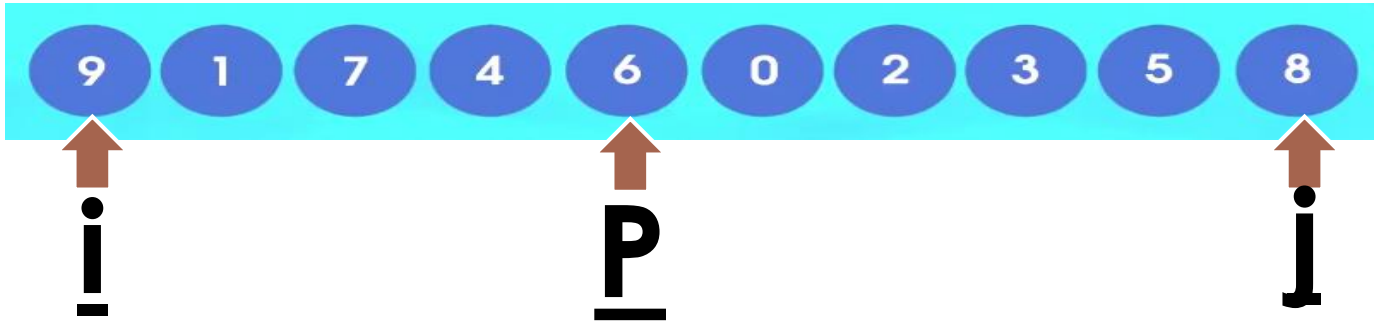


Método Quicksort

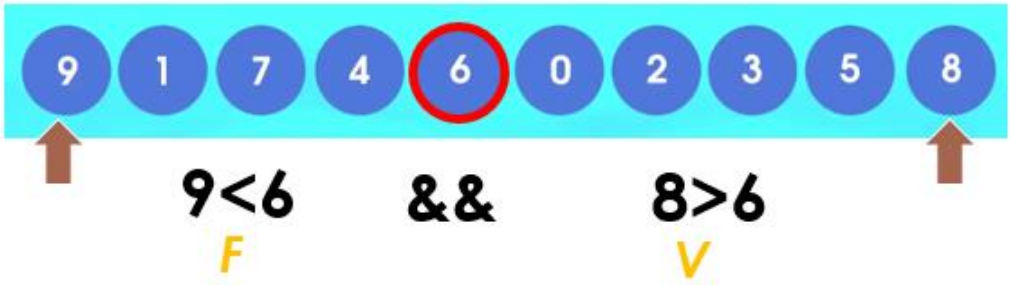
Método Quicksort

PIVOTE como elemento **central**.

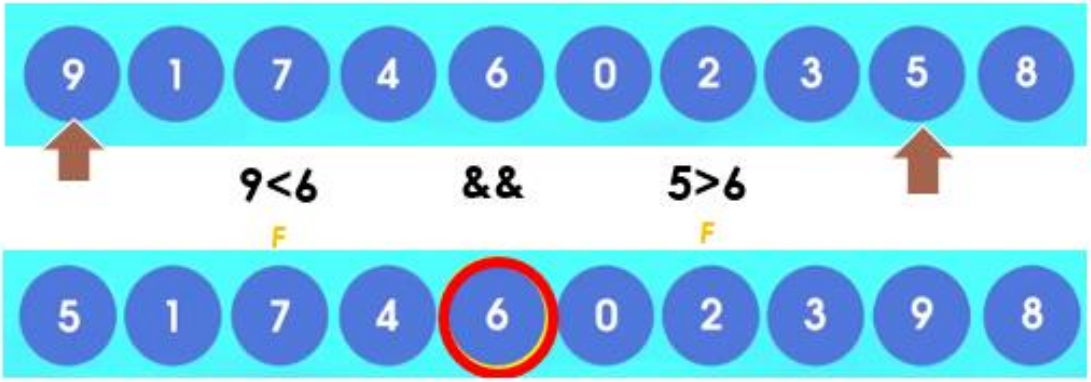
- Elementos menores a la derecha y mayores a la izquierda.
- Se requiere dos punteros (inicio – fin).



Método Quicksort



- Recorrer el puntero del lado derecho, para hacer otra comparación.

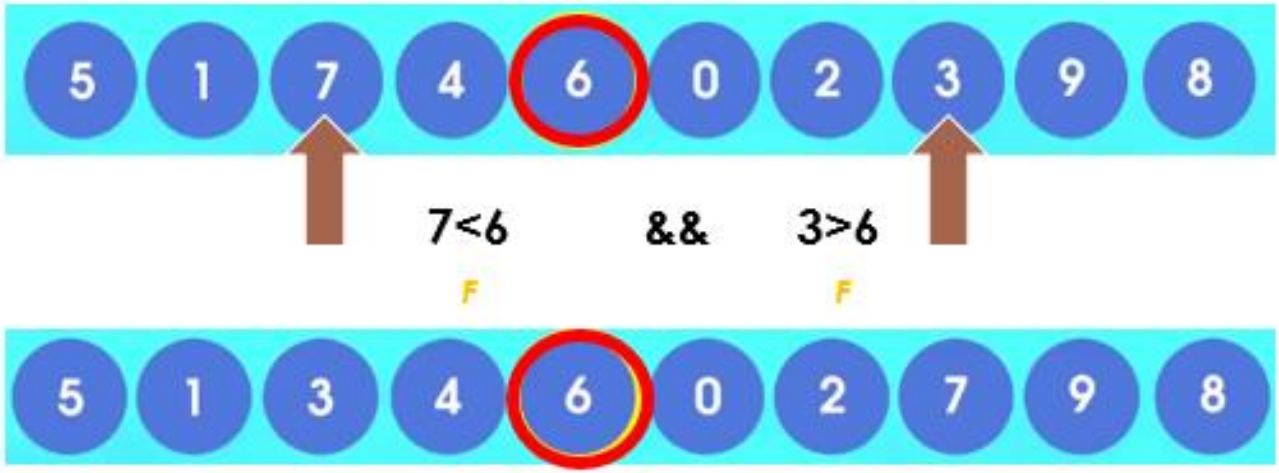


Método Quicksort

- Nuevamente recorrer los punteros, para hacer la comparación.

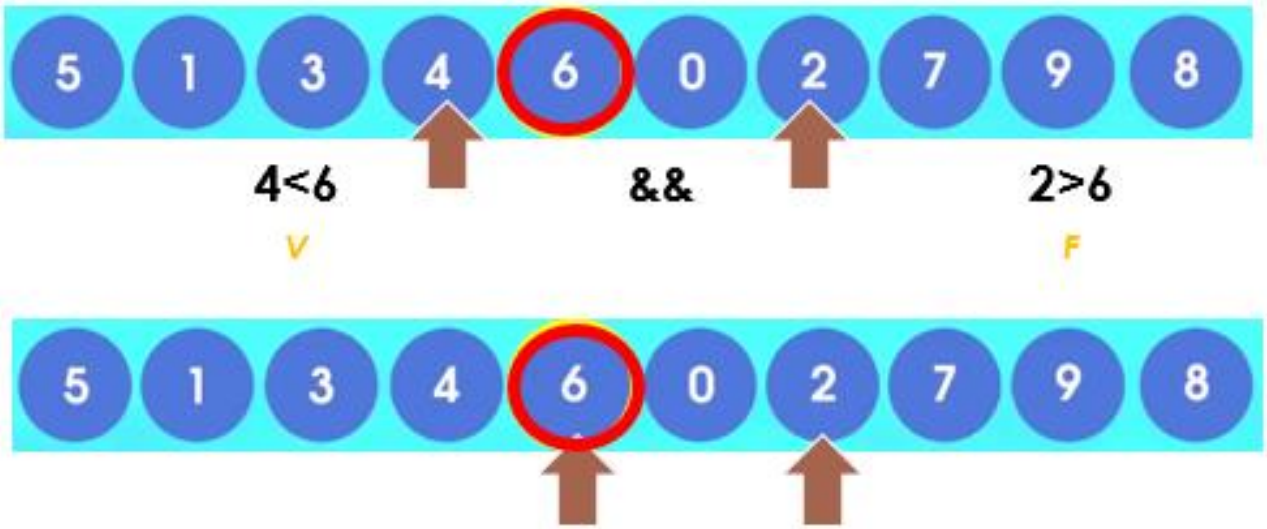
Método Quicksort

- Se procede otra vez a comparar.



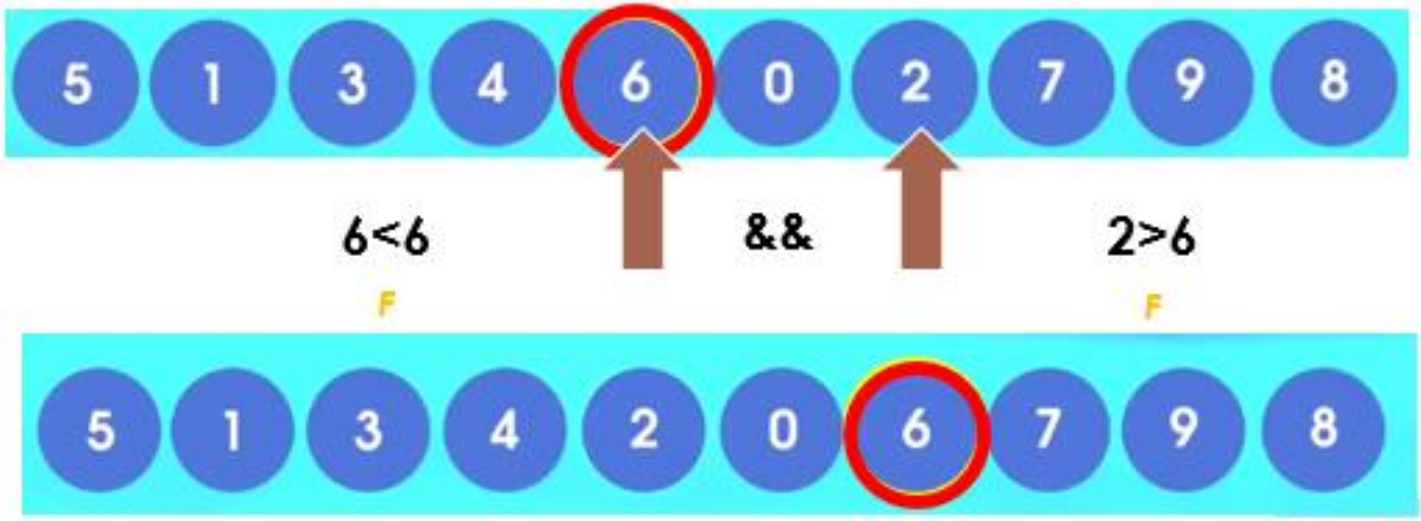
Método Quicksort

- Recorrer el puntero del lado derecho, para hacer otra comparación.



Método Quicksort

- Se procede otra vez a comparar.

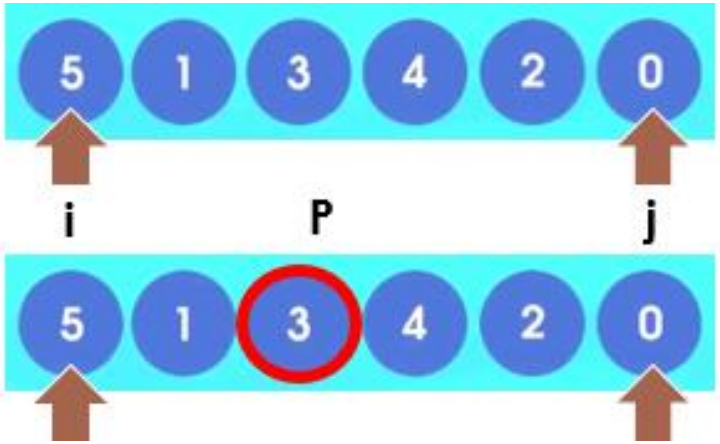


Método Quicksort

- Los 2 sub-arreglos se procede a hacer otra vez el mismo proceso.

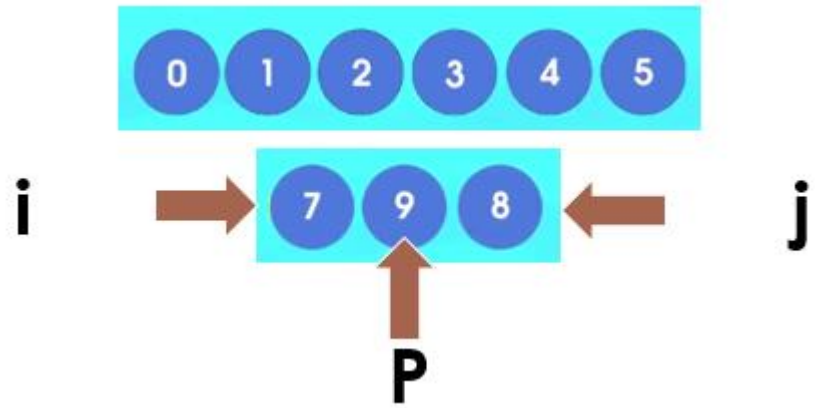
Método Quicksort

Elegir un elemento de la lista de elementos a ordenar, al que llamaremos ***PIVOTE***. De la misma 2 punteros que van a recorres las posiciones del arreglo



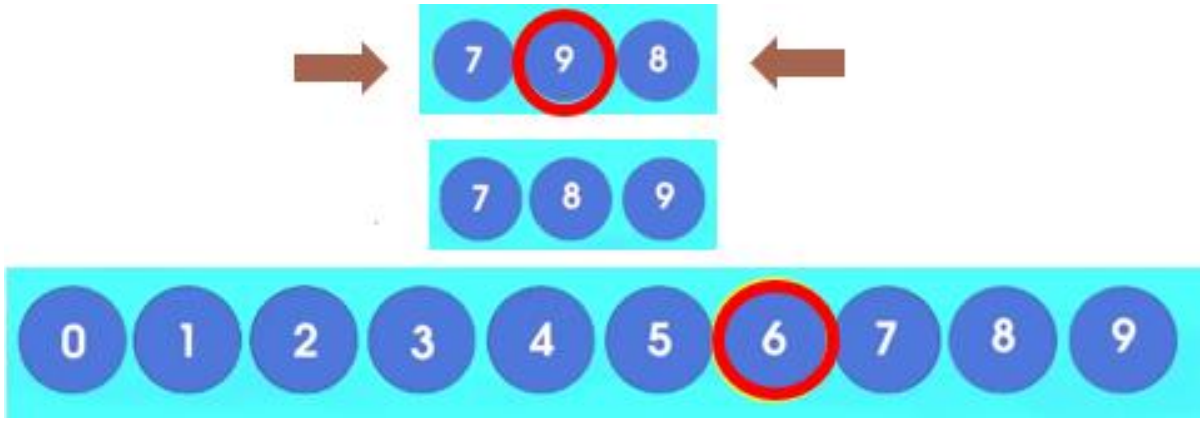
Método Quicksort

Hasta que todos los elementos queden ordenados



Método Quicksort

Dejar los elementos **MENORES** del pivote a la izquierda y los números **MAYORES** del pivote a la derecha



Método Quicksort

- La eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.
- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño.
- En el peor caso, el pivote termina en un extremo de la lista.
- El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas.

6 5 3 1 8 7 2 4

Algoritmos de Ordenación Interna

```
Algoritmo Quicksort(a[], primero, ultimo)
   si primero < ultimo entonces
      int i,j,central
      double pivote
      central = (primero + ultimo) / 2
      pivote = a[central]
      i = primero
      j = ultimo
      repetir
         mientras a[i] < pivote hacer
            i = i + 1
         mientras a[j] > pivote hacer
            j = j - 1
         si i <= j entonces
            intercambiar(a[i], a[j])
            i = i + 1
            j = j - 1
      hasta que i > j
      si primero < j entonces
         Quicksort(a, primero, j)  // mismo proceso con sublista izquierda
      si i < ultimo entonces
         Quicksort(a, i, ultimo) // mismo proceso con sublista derecha
Fin Algoritmo
```

Método Quicksort

Método Quicksort (en parejas)

PIVOTE como elemento **inicial**.

- Elementos menores a la derecha y mayores a la izquierda.

10	3	6	9	1	4	2	5	7	8

Puntuación Extra!!

Lección 2.1.4 : ShellSort

Ordenación por inserción con incrementos decrecientes (ShellSort)

- Cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.
- Realiza saltos de mayor tamaño y con ello consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial **n/2**, luego en cada iteración se reduce el salto a la mitad, hasta que el salto es de tamaño 1.

Método ShellSort

1. Se divide la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.
2. Se clasifica cada grupo por separado, comparando las parejas de elementos y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos (n/4), con un salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.
4. El algoritmo termina cuando el tamaño del salto es 1.

Método ShellSort

En el canal de YouTube de Hugo López encontramos la descripción del método ShellSort

https://www.youtube.com/watch?v=bJ-LWnpyx6s

Método ShellSort

- Primero dividimos el arreglo original a la mitad (n/2).
- El valor resultante corresponde a la distancia entre los elementos.

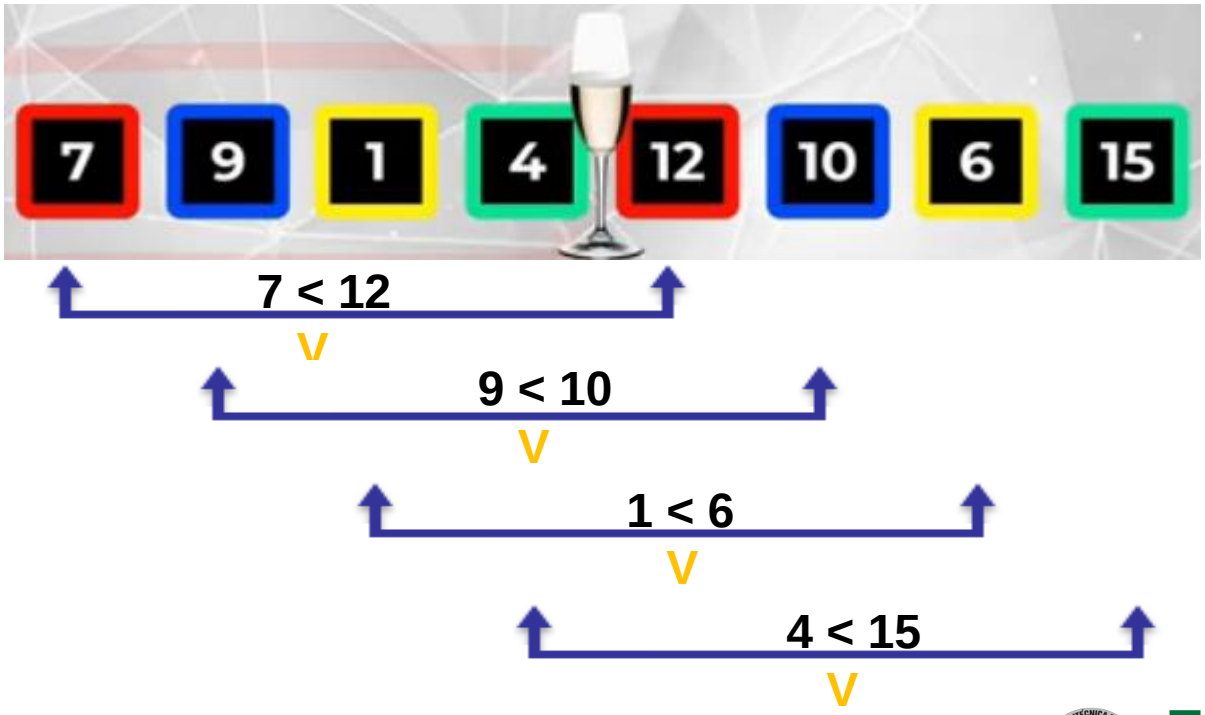
Distancia = 8/2 → 4

Método ShellSort

- SI los elementos no fuesen numero par por ejemplo 7/2=3.5
- Tomamos los 3 elementos a la izquierda, y los 4 a la derecha (sobrando 1)

Método ShellSort

- Comparar los elementos de cada grupo.



Método ShellSort

- Dividimos nuevamente los elementos.

Distancia = 2/2 → 2

Método ShellSort

7 < 1

F

7 < 12

V

12 < 6

F

Método ShellSort

9 < 4

F

- Hacemos el mismo proceso con el color azul.

9 < 10

V

10 < 15

V

Método ShellSort

1 < 7

V

- Hacemos el mismo proceso con el color rojo (nuevamente).

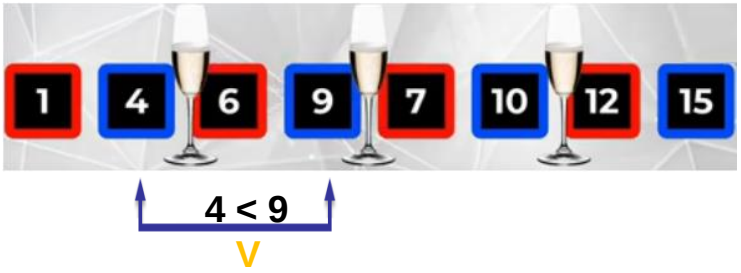
7 < 6

F

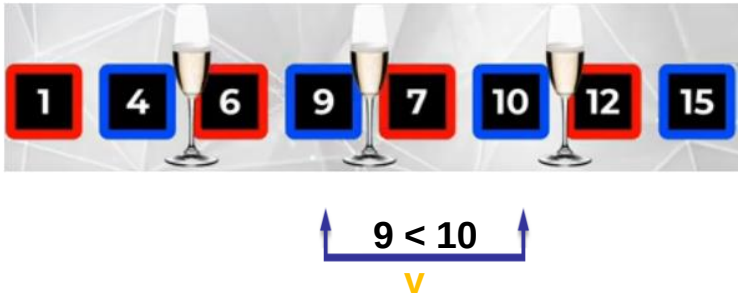
7 < 12

V

Método ShellSort



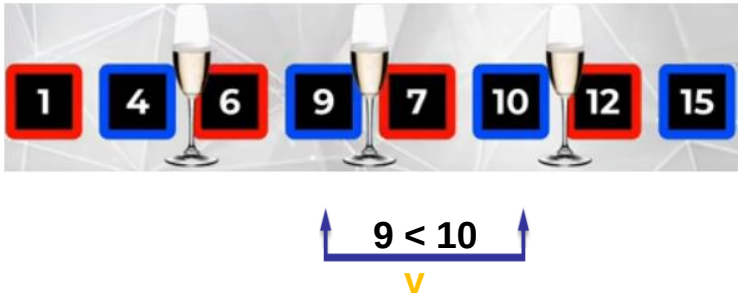
- Hacemos el mismo proceso con el color azul (nuevamente).



- No hay cambios "***Elementos Ordenados***" por parte del ***azul***.

Método ShellSort

- Hacemos el mismo proceso con el color azul (nuevamente).



- No hay cambios para el proceso.

Método ShellSort

- Si realizamos el proceso nuevamente no habrán mas cambios.
- Siguiente paso dividir la distancia.

Distancia = 2/2 → 1

Método ShellSort

< < < <

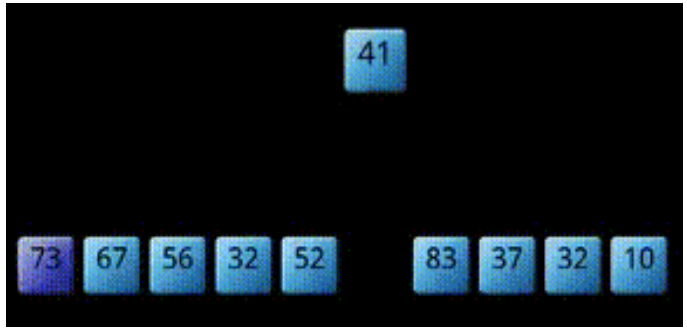
V V V F (intercambio)

Método ShellSort

- Elementos ordenados

Método ShellSort

```
Algoritmo Shellsort(a[], n)
Salto = n / 2
mientras (salto > 0) hacer
   //para dividir la lista en grupos y clasificar cada grupo se anida este código
   para i = salto hasta n hacer
         j = i - salto
         mientras (j >= 0) hacer
              k = j + salto
              si (a[j] <= a[k]) entonces
                   j = -1
              sino
                   Intercambio (a[j], a[j+1])
                   j = j - salto
              fin_si
         fin_mientras
   fin_para
fin_mientras
Fin Algoritmo
```



Lección 2.1.5 :
Ordenación por Distribución

Ordenación por distribución

- Ordenan el arreglo tomando cada número e insertándolo en la posición que toma su valor, es decir, si se tiene un cinco se coloca en la posición cinco del arreglo, algo así como: “lo que valgas en esa posición vas”.
- No se podrán ordenar los arreglos que tengan valores repetidos sin incrementar la capacidad de la posición.
- El tamaño del arreglo, será el número más grande que se encuentre en él.
- Algoritmos de ordenamiento po distribución tenemos:
  - BucketSort
  - RadixSort

Lección 2.1.5.1: BucketSort o BinSort

Ordenamiento por casilleros (BucketSort)

- Distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero solo puede contener los elementos que cumplan unas determinadas condiciones.
- Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos.
- Después cada uno de esos casilleros se ordena individualmente con otro algoritmo de ordenación o se aplica recursivamente el algoritmo para obtener casilleros con menos elementos.

Método BucketSort

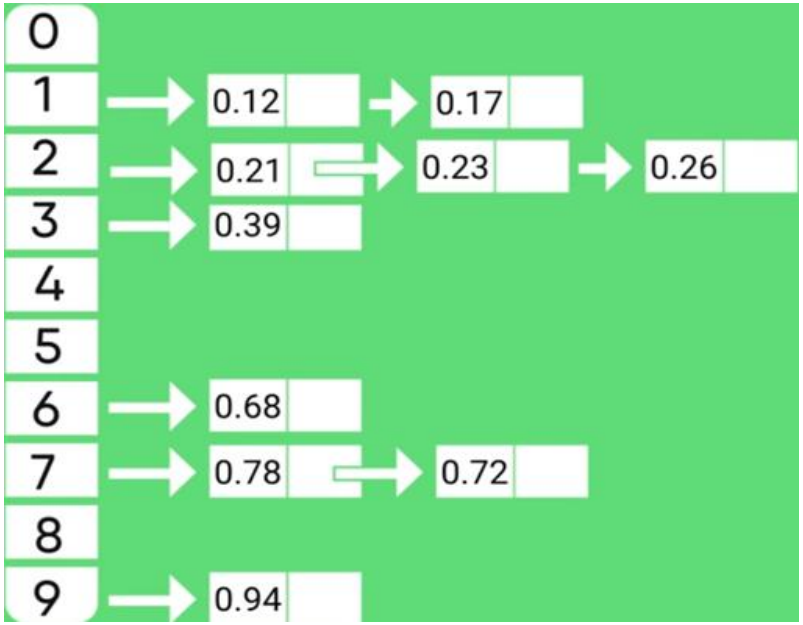
1. Crear una colección de casilleros vacíos.
2. Colocar cada elemento a ordenar en un único casillero.
3. Ordenar individualmente cada casillero.
4. Devolver los elementos de cada casillero concatenados por orden.

Método BucketSort

En el canal de YouTube de Hugo López encontramos la descripción del método BucketSort (forma 1)

https://www.youtube.com/watch?v=VuXbEb5ywrU

Método BucketSort



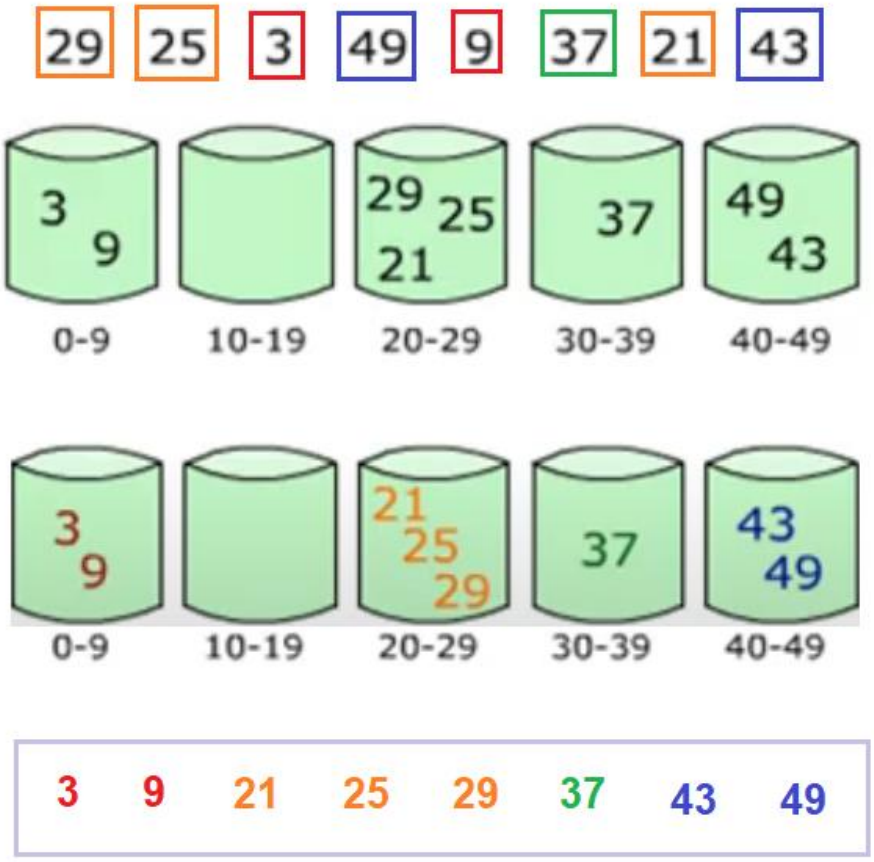
0.12	0.17	0.21	0.23	0.26	0.39	0.68	0.72	0.78	0.94

Método BucketSort

En el canal de YouTube de Hugo López encontramos la descripción del método BucketSort (forma 2)

https://youtu.be/m5iEuSaNlDI?t=218s

Método BucketSort



Algoritmos de Ordenación Interna

```
Algoritmo bucketSort(vector, n)
CrearUrnas(Urnas);
desde j = 0 hasta n hacer
      agregoEnUrna(Urnas[vector[j].clave], vector[j]);
fin_desde
i =1;
mientras EsVacia(Urnas[i]) hacer
   i = i+1
fin_mientras
desde j = i+l a m hacer
   EnlazarUrna(Urnas[i], Urnas[j]);
fin_desde
j = 1;
dir = <frente Urnas[i]>;
mientras dir <> nulo hacer
   vector[j] = < elemento apuntado por dir>;
   j = j+i;
   dir = Sgte(dir)
fin_mientras
Fin Algoritmo
```

Método BucketSort

Lección 2.1.5.2: Ordenación por Radix

Ordenamiento por residuos (RadixSort)

- Consiste en ordenar los números tomando en cuenta el valor relativo que tienen los dígitos de un número en un determinado sistema de numeración (decenas, centenas).
- La característica de este algoritmo está en que no hace comparaciones para ordenar las listas, simplemente se encarga de ir contando o agrupando los números que tengan el mismo valor relativo en determinada cifra.
- Se pueden ir agrupando o contando los números, desde las cifras menos significativas a las más significativas o viceversa.
- Complejidad algorítmica O(nk), para todos los casos.

Método RadixSort

1. Recorrer los elementos de la colección.
2. Insertar en el index del array auxiliar el correspondiente al digito revisado.
3. Extraer los elementos del array y colocarlo de nuevo en la colección.
4. Realizar los mismos pasos con el siguiente dígito significativo.

Método RadixSort

En el canal de YouTube de Hugo López encontramos la descripción del método RadixSort

https://www.youtube.com/watch?v=Om4BljCs_qE

Método RadixSort

- Ordena en base al 1er digito.
- Se crea un arreglo con índices desde el 0 al 9.

Método RadixSort

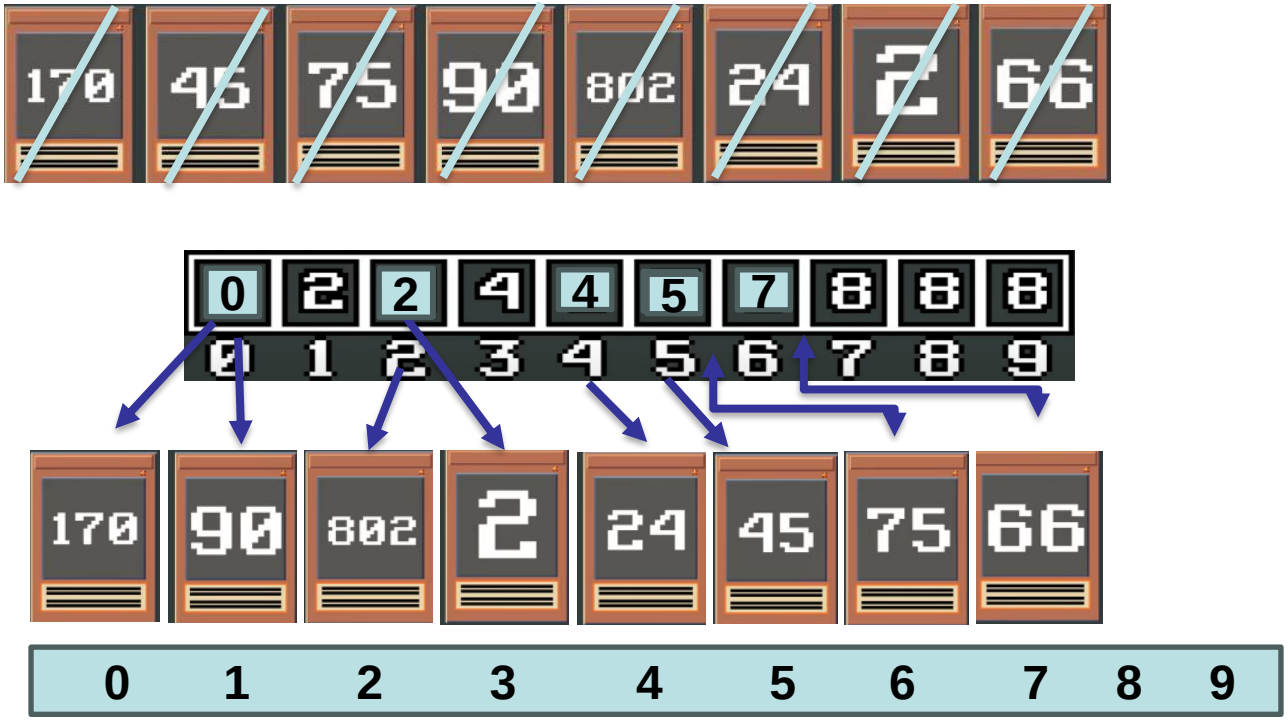
- Contamos acorde al digito (**unidades**).

Método RadixSort

- Realizamos una suma acumulativa.

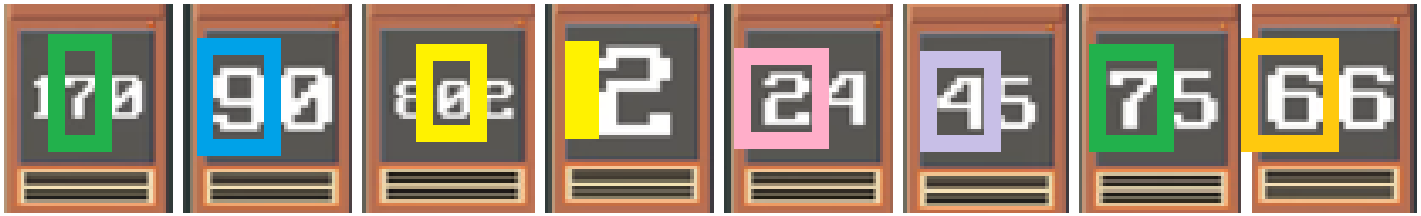
Método RadixSort

- Colocamos las elementos en su posición.



Método RadixSort

- Contamos acorde al digito (**decena**).

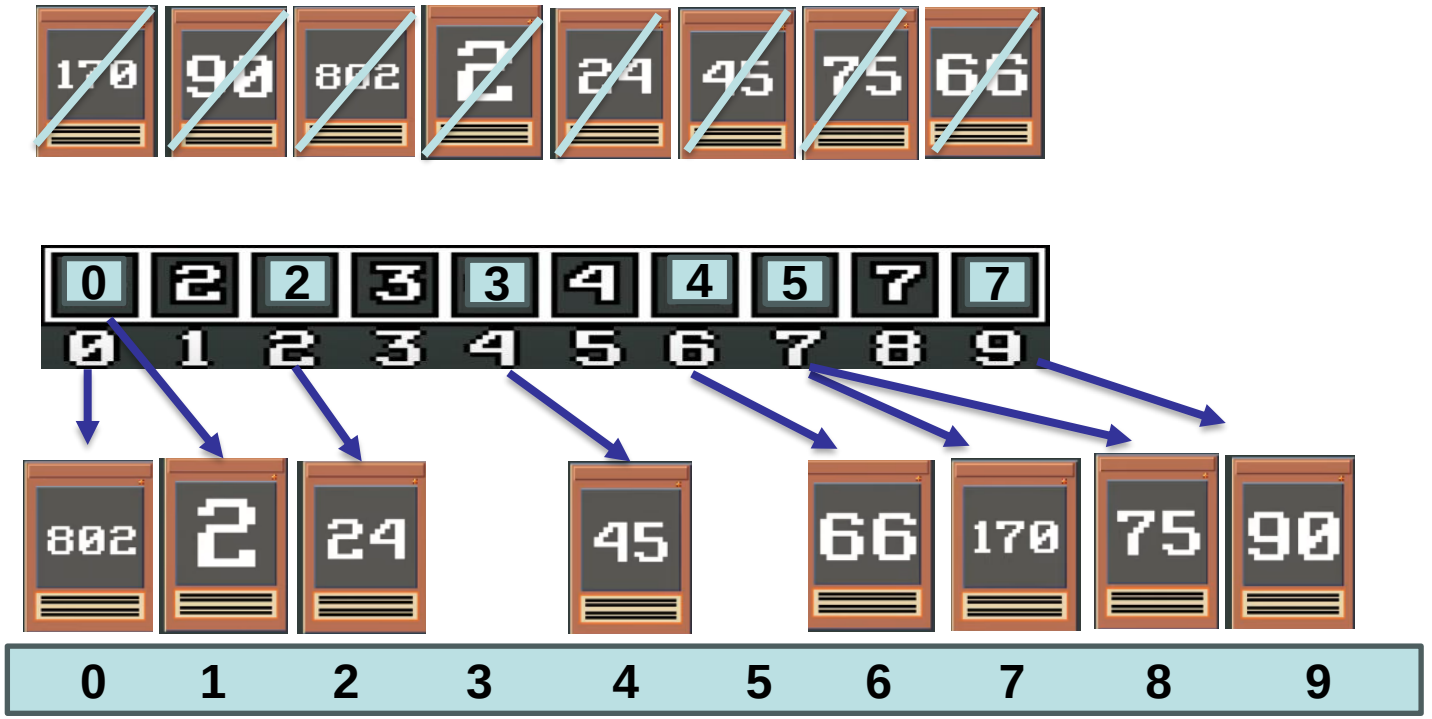


Método RadixSort

- Realizamos una suma acumulativa.

Método RadixSort

- Colocamos los elementos en su posición.

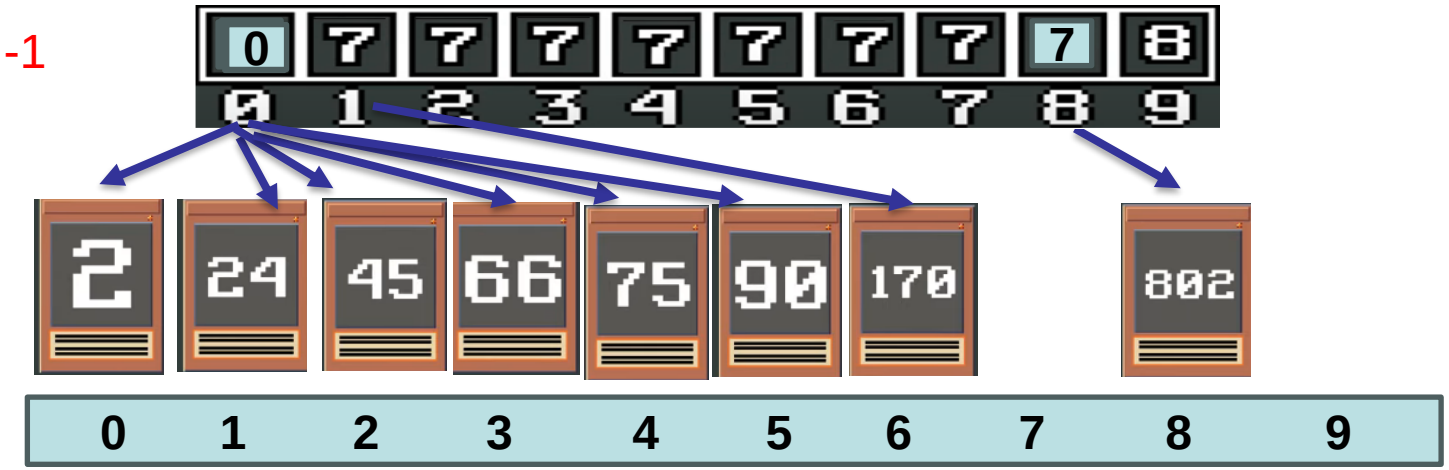


Método RadixSort

- Contamos acorde al digito (**centenas**).
- Realizamos una suma acumulativa

Método RadixSort

- Colocamos los elementos en su posición.



Algoritmos de Ordenación Interna

```
Algoritmo radixSort(vector, n)
max1 = max(arr)
exp = 1
mientras max1 / exp >= 1 hacer
    countingSort(arr,exp) //realizar algoritmo
    exp *= 10
Fin mientras
Fin Algoritmo
```

Método RadixSort

Algoritmos de Ordenación Interna

Algoritmo radixSort(vector, n)
max1 = max(arr)
exp = 1
mientras max1 / exp >= 1 hacer
    countingSort(arr,exp)
    exp *= 10
Fin Algoritmo

```python
def countingSort(arr, exp1):
    n = len(arr)
    output = [0] * (n)
    count = [0] * (10)

    for i in range(0, n):
        index = arr[i] // exp1
        count[index % 10] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    i = n - 1
    while i >= 0:
        index = arr[i] // exp1
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    i = 0
    for i in range(0, len(arr)):
        arr[i] = output[i]
```

Método
RadixSort

Preguntas ?