



# ALGORITMOS DE ORDENAMIENTO Y BÚSQUEDA

# índice



01

Algoritmos  
de Búsqueda



02

Algoritmos de  
Ordenamiento





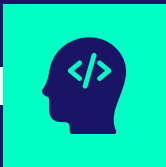
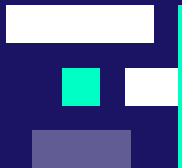
01

# ALGORITMOS DE BÚSQUEDA

# introducción

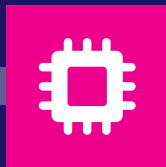
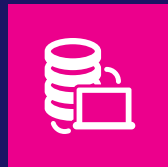
- Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento concreto dentro de una estructura de datos para entregar una respuesta exacta.
- El objetivo es crear una función que hará lo siguiente: Dada una lista de números, tenemos que verificar si un número "n" está dentro de esta lista; además, guardaremos el número de pasos que nos tomó obtener el resultado.
- Existen diferentes algoritmos de búsqueda y la elección depende de la forma en que se encuentren organizados los datos: si se encuentran ordenados o si se ignora su disposición o se sabe que están al azar.

# 1.1 Búsqueda secuencial



Más simple, menos eficiente y que menos precondiciones requiere: no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio.

Se utiliza cuando el contenido del Vector no se encuentra o no puede ser ordenado.



compara cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.

Este es el método de búsqueda más lento, este es el único método cuando nuestros datos están desordenados.



```

1
2 def busqueda_lineal(array, n, x):
3
4     for i in range(0, len(array)): #recorre el array de inicio a fin
5         if (array[i] == x): #condicional de comparacion
6             return i
7     return -1
8
9 array = [10, 8, 3, 1, 2, 7, 0]
10 print("\narray= ",array,"\n")
11 n = len(array)
12 x = int(input("Ingrese el elemento a buscar en el arary: "))
13
14 index = busqueda_lineal(array, len(array), x)
15 if(index == -1):
16     print("Elemento no encontrado !!!")
17 else:
18     print("El elemento << {} >> se encuentra en el indice: {}".format(x,index))
19

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS D:\LABORATORIO\_ESTRUCTURA\_DATOS> & C:/Users/USUARIO/AppData/Local/Programs/Python/Python37/python.exe c

array= [10, 8, 3, 1, 2, 7, 0]

Ingrese el elemento a buscar en el arary: 7

El elemento << 7 >> se encuentra en el indice: 5

# Complejidad De La Búsqueda Secuencial

## Mejor Caso

El algoritmo de búsqueda lineal termina tan pronto como encuentra el elemento buscado en el array



$O(1)$

## Peor Caso

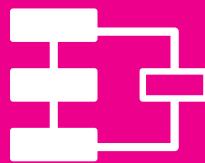
Sucede cuando encontramos X en la última posición del array.



$O(an+b) = O(n)$ .

## Caso Medio

Supongamos que cada elemento almacenado en el array es igualmente probable de ser buscado.



Media =  $(\text{Total} / n) = a((n+1) / 2) + b$  que es  $O(n)$ .

# Ventaja de la Búsqueda Secuencial

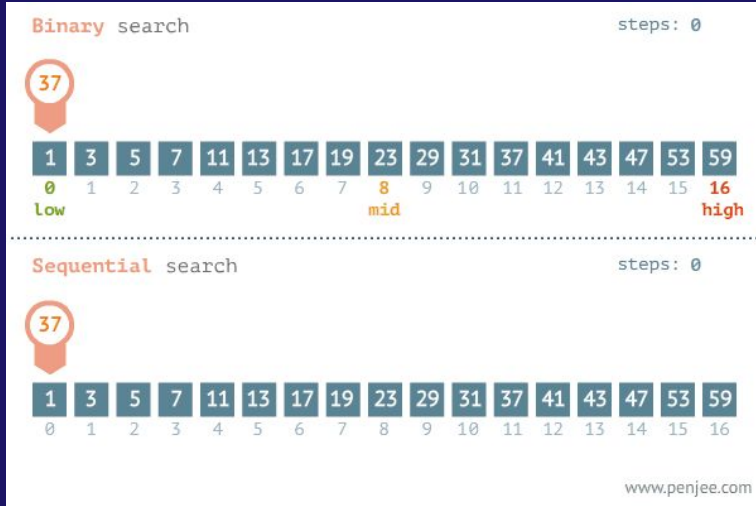
- Es el algoritmo más simple de búsqueda y no requiere ningún proceso previo de la tabla, ni ningún conocimiento sobre la distribución de las llaves.
- La búsqueda secuencial es el área del problema donde previamente existían mejores algoritmos.
- Único método de búsqueda para datos no ordenados.



# Desventajas de la Búsqueda Secuencial

- Este método de búsqueda es muy lento, pero si los datos no están en orden es el único método que puede emplearse para hacer las búsquedas.
- Si los valores de la llave no son únicos, se requiere buscar en toda la lista.
- Si los registros a los que se accede con frecuencia no están al principio del archivo, la cantidad promedio de comparaciones aumenta notablemente.
- Para las aplicaciones interactivas que incluyen peticiones o actualizaciones de registros individuales, los archivos secuenciales ofrecen un rendimiento pobre.

# BÚSQUEDA BINARIA



Divide el array en mitades,  
es por eso el nombre  
“binaria”, para que la  
búsqueda sea más sencilla.

# BÚSQUEDA BINARIA

Algoritmo:

1. Verifica que la lista tenga elementos. Si no los hay termina.
2. Compara el elemento a buscar con el número que se encuentra a la **mitad** de la lista.
3. Si es el que se busca regresa el número y termina.
4. Si no es el número buscado:
  - a. Si el número es menor al elemento de la mitad de la lista realiza la búsqueda binaria con la sublista izquierda.
  - b. Si no realiza la búsqueda binaria con la sublista derecha.



# BÚSQUEDA BINARIA

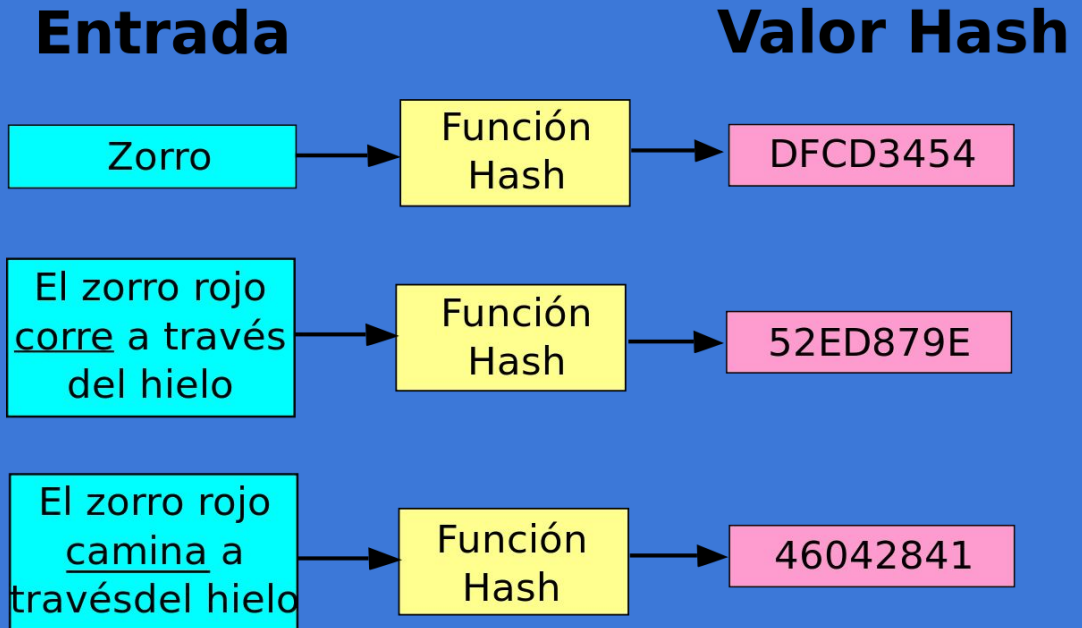
## Ejemplo con python

```
def binaria(lista, x):  
    if len(lista) <= 0:  
        return "Número no encontrado"  
  
    m = lista[len(lista)//2]  
    if m == x:  
        return "Número encontrado"  
    else:  
        if x < m:  
            return binaria(lista[:len(lista)//2], x)  
        else:  
            return binaria(lista[(len(lista)//2)+1:], x)  
  
lista = [-5, -1, 2, 4, 5, 6, 10, 27]  
print(binaria(lista, 4))
```

➞ Número encontrado

# BÚSQUEDA HASHING

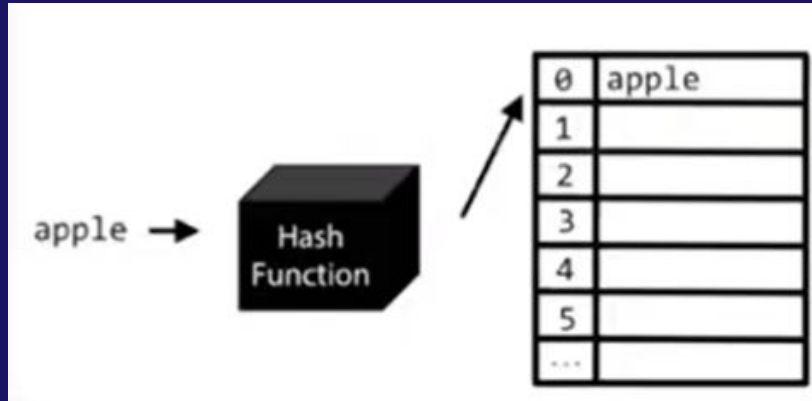
Son funciones que se crean, a partir de un dato de entrada, una salida finita que normalmente es una cadena de longitud fija. esta cadena representa un "resumen" de la información de entrada



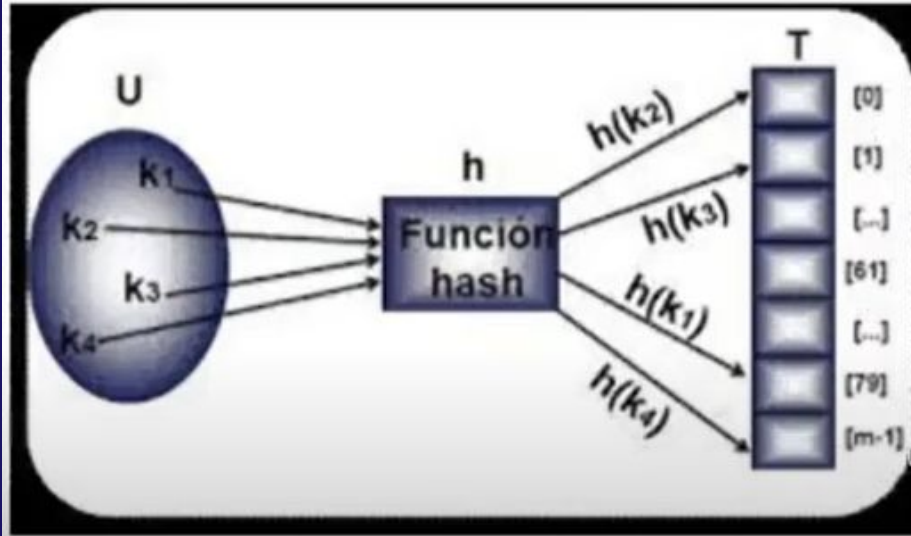
# BÚSQUEDA HASHING

## Tabla Hash

Una tabla hash es una estructura que asocia claves con valores, La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos almacenados a partir de una clave generada usando el nombre, número de cuenta o id.



# BÚSQUEDA HASHING



Función hash:

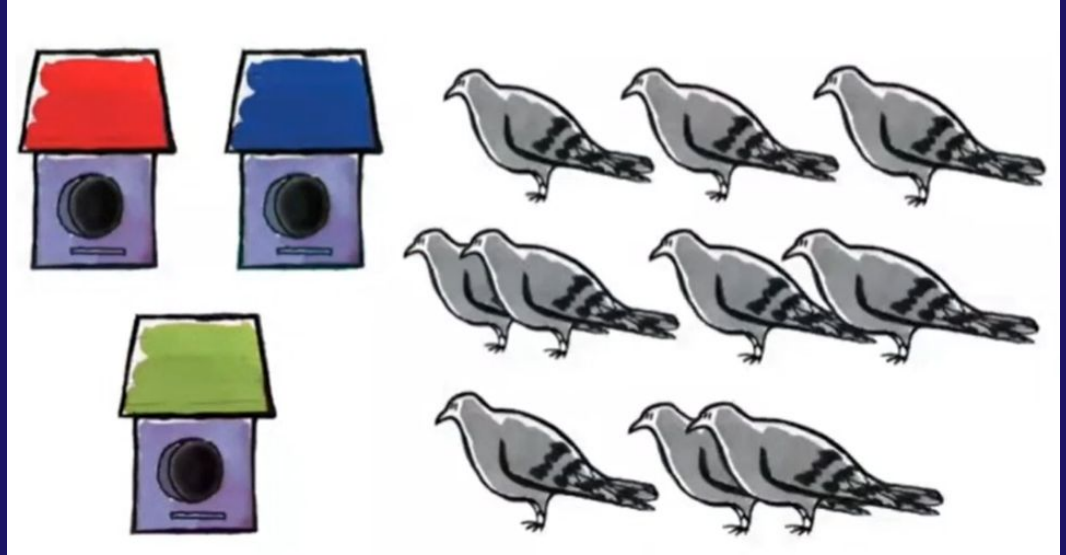
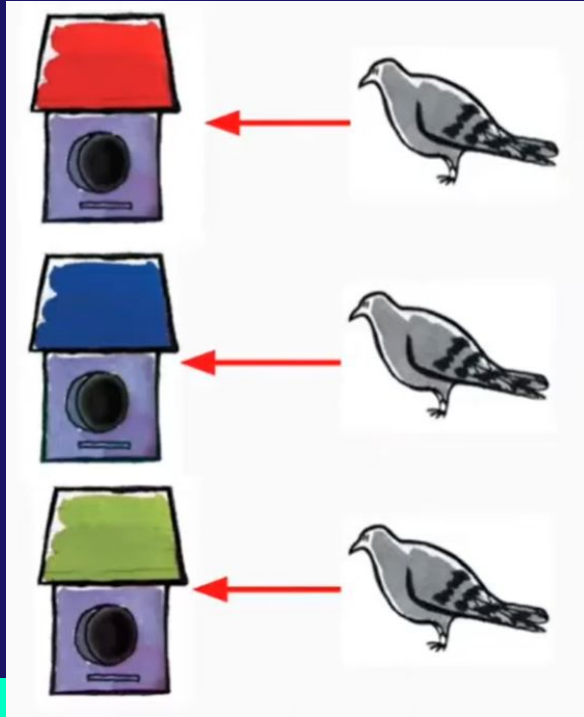
$$h(k) = k \% m$$

donde:

- $k$ : elemento del que obtendremos su valor hash
- $m$ : un número igual al tamaño de la tabla hash

# BÚSQUEDA HASHING

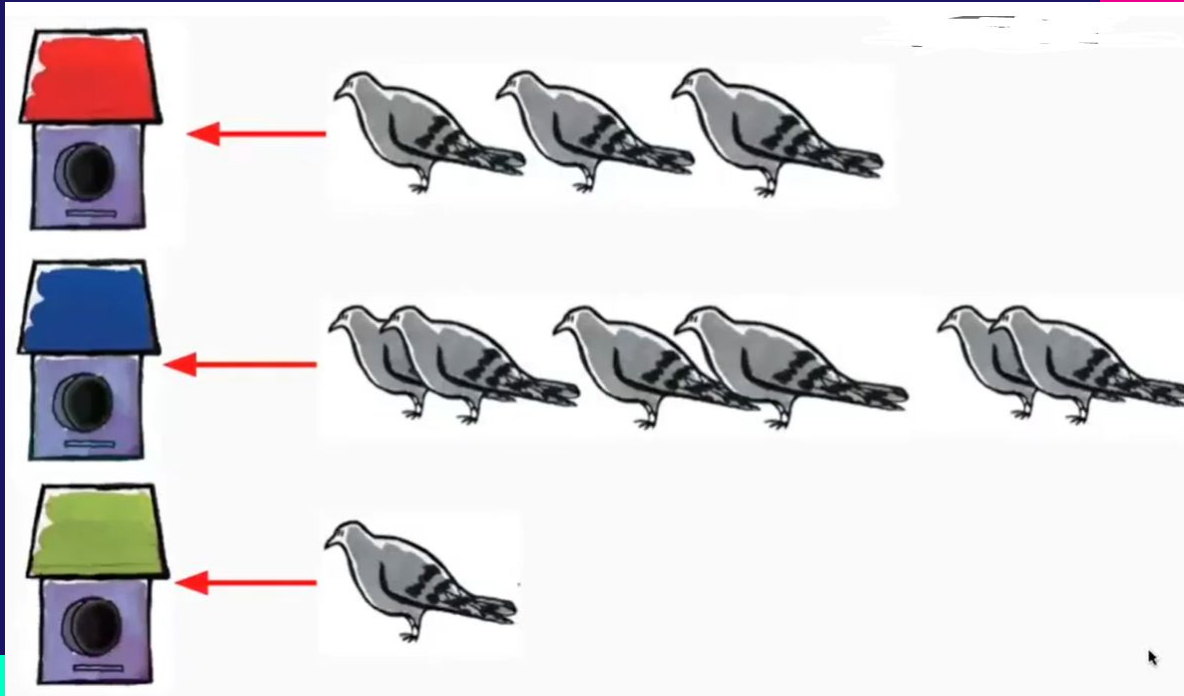
## Principio del Palomar





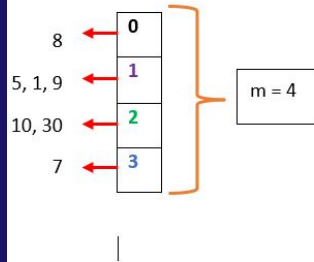
# BÚSQUEDA HASHING

## Principio del Palomar



# BÚSQUEDA HASHING

## Ejemplo



$K \% m$

donde:

k: elemento del que obtendremos su valor hash.

m: número igual al tamaño de la tabla hash (longitud)

$$5 \% 4 = 1$$

$$7 \% 4 = 3$$

$$1 \% 4 = 1$$

$$8 \% 4 = 0$$

$$10 \% 4 = 2$$

$$30 \% 4 = 2$$

$$9 \% 4 = 1$$

Números a ingresar :

5, 7, 1, 8, 10, 30, 9

Elemento a buscar:

1, 8

# BÚSQUEDA HASHING

## Ejemplo con python

```
def convCad(cad):
    salida = ""
    for j in cad:
        salida += str(ord(j))
    return int(salida)
def hashM(cad,m):
    i = convCad(cad)
    return int(m*(i * 0.000000000000212324 % 1))
def agregar(cad,ht,m):
    res=hashM(cad,m)
    ht[res].append(cad)
def buscar(cad,ht,m):
    h = hashM(cad,m)
    for i in ht[h]:
        if i == cad:
            return True
    return False
```

```
m=19
ht = [[] for i in range(m)]
agregar("pizarron", ht,m)
agregar("plumon", ht,m)
agregar("borrador", ht,m)
agregar("impresora", ht,m)
agregar("pluma", ht,m)
agregar("cuaderno", ht,m)
print(ht)
print(buscar("pluma", ht,m))
print(buscar("libro", ht,m))
```

```
[[], [], [], [], ['plumon'], [], ['borrador'], ['pluma'], ['impresora'], ['cuaderno'], [], [], [], [], [], [], [], ['pizarron']]
True
False
```

# ORDENAMIENTO DE ALGORITMOS



# Algoritmos de Ordenamiento

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar información de una manera especial basándonos en un criterio de ordenamiento.

Los algoritmos de ordenación sirven para reorganizar el orden de los elementos de una estructura, como un vector. Muestran de forma estructurada cómo ordenar los elementos que haya dentro de una estructura, como un array.

Una de las características de los algoritmos de ordenación es que se puede ordenar cualquier estructura con elementos que sean ordenables.

- Podemos ordenar números, porque unos son mayores que otros
- Podemos ordenar meses, porque unos vienen antes que otros.

# Ordenamiento por Burbuja

Este algoritmo de clasificación simple itera sobre la lista de datos, comparando elementos en pares hasta que los elementos más grandes “**burbujean**” hasta el final de la lista y los más pequeños permanecen al principio.

## Ordena

Valores de una lista

## Busca

el menor elemento de la lista e intercambiarlo con el primero

## Ventajas

Los elementos se intercambian sin utilizar almacenamiento temporal adicional, de modo que el espacio requerido es el mínimo.

## Desventajas

No se recomienda utilizarlo ya que consta de un código bastante extenso y no se comporta adecuadamente con una lista

## Pasos para el ordenamiento por selección

6

4

2

5

3

1

1

2

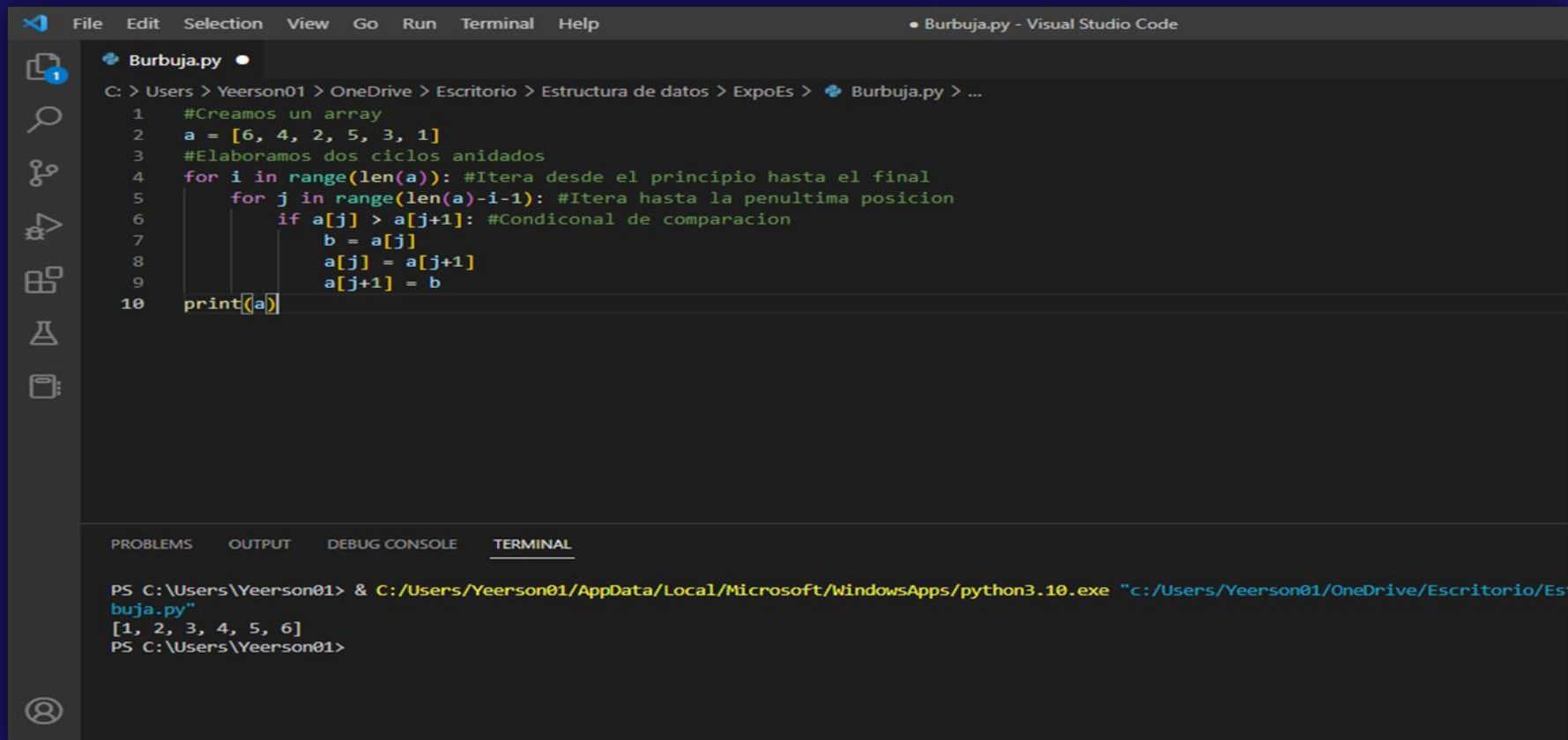
3

4

5

6

# Codigo



The image shows a Visual Studio Code window with a Python file named 'Burbuja.py' open. The code implements a bubble sort algorithm. The terminal at the bottom shows the command to run the script and the output, which is the sorted array [1, 2, 3, 4, 5, 6].

```
File Edit Selection View Go Run Terminal Help
Burbuja.py
C: > Users > Yeerson01 > OneDrive > Escritorio > Estructura de datos > ExpoEs > Burbuja.py > ...
1  #Creamos un array
2  a = [6, 4, 2, 5, 3, 1]
3  #Elaboramos dos ciclos anidados
4  for i in range(len(a)): #Itera desde el principio hasta el final
5      for j in range(len(a)-i-1): #Itera hasta la penultima posicion
6          if a[j] > a[j+1]: #Condiconal de comparacion
7              b = a[j]
8              a[j] = a[j+1]
9              a[j+1] = b
10 print(a)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Yeerson01> & C:/Users/Yeerson01/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/Yeerson01/OneDrive/Escritorio/Es-
buja.py"
[1, 2, 3, 4, 5, 6]
PS C:\Users\Yeerson01>
```



# Ordenamiento por Inserción

Este algoritmo, consiste en el recorrido por la lista seleccionado en cada iteración un valor como clave y compararlo con el resto insertandolo en el lugar correspondiente

## Ordena

Valores de una lista

## Busca

el menor elemento de la lista e intercambiarlo con el primero

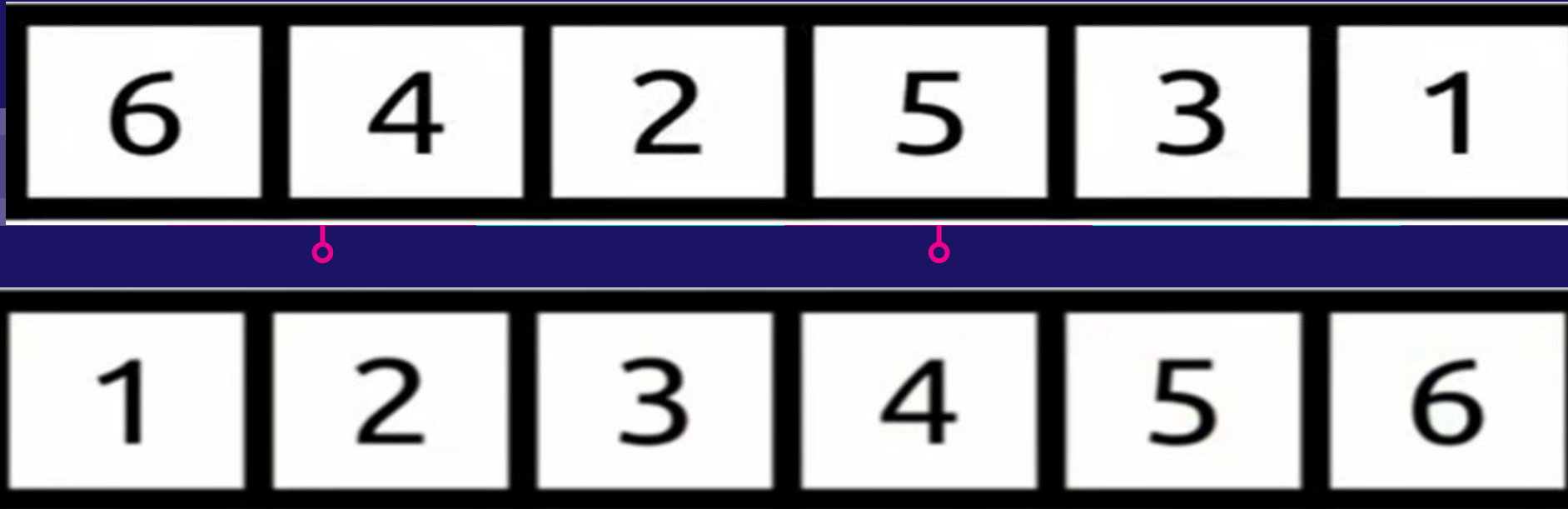
## Ventajas

La principal ventaja es su simplicidad y buen rendimiento cuando se trabaja con una pequeña lista.

## Desventajas

No funciona bien con una lista grande. Por lo tanto, este solo es útil cuando se ordena una lista de pocos elementos.

## Pasos para el ordenamiento por Inserción



# Codigo

Burbuja.py • Selecion.py Insercion.py X Ordenacion.py

C: > Users > Yeerson01 > OneDrive > Escritorio > Estructura de datos > ExpoEs > Insercion.py > ...

```
1  #creamos un array
2  a = [6, 4, 2, 5, 3, 1]
3  #Itera todos los elemntos del arreglo
4  for i in range(0, len(a)):
5      minimo = i #Al inici guaradmos nuestro dato minimo
6      for j in range(i+1, len(a)): #Itera el dato minimo+1 hasta el final
7          if a[minimo] > a[j]: #Hacemos la comparacion del minimo y el j
8              minimo = j
9
10     b = a[i] #Intercambiamos las asignaciones
11     a[i] = a[minimo]
12     a[minimo] = b
13 print(a)
14
15
```

PROBLEMS

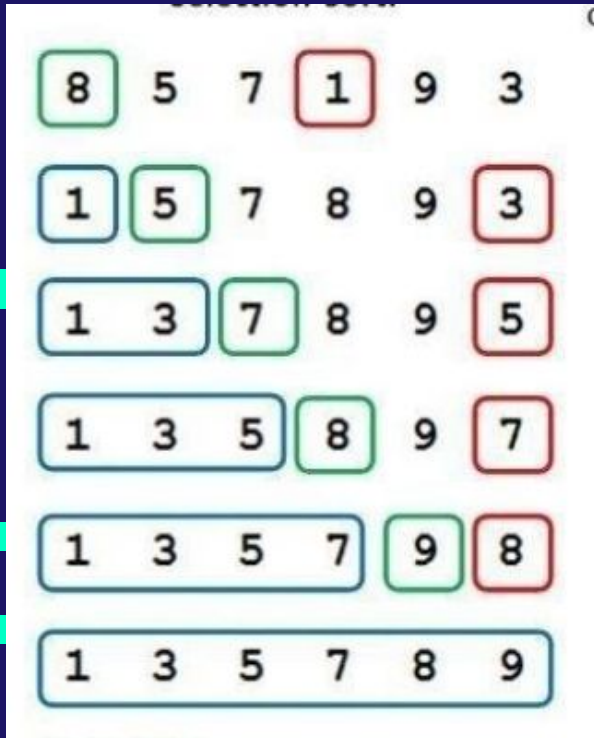
OUTPUT

DEBUG CONSOLE

TERMINAL

```
buja.py"
[1, 2, 3, 4, 5, 6]
ercion.py"
```

# Ordenamiento por Selección



## Ordena

Valores de una lista

## Busca

el menor elemento de la lista e intercambiarlo con el primero

## Ventajas

Eficiente con lista pequeña. No hay almacenamiento temporal adicional

## Desventajas

poca eficiencia cuando se trata con una enorme lista de elementos.

# Pasos para el ordenamiento por selección

buscamos al mínimo  
elemento de la lista y lo  
llevamos adelante

`lista = [3, 1, 4, 2]`

`lista = [1, 3, 4, 2]`

`lista = [1, 2, 4, 3]`

`lista = [1, 2, 3, 4]`

TENEMOS UNA LISTA

# Código

```
#Ordenamiento por selección
def ordenSeleccion(lista):

    for i in range(len(lista)-1): #bucle padre inicio i=0
        min = i #primer elemento será el minimo
        #Ciclo for anidado
        for j in range(i+1,len(lista)): #bucle hijo el que busca al numero menor
            if lista[j]<lista[min]:
                min=j

        if min != i:
            lista[min], lista[i] = lista[i], lista[min]

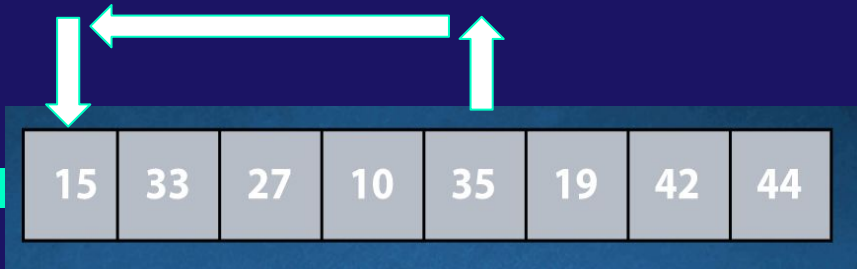
def mostrarLista(lista):
    for numero in lista:
        print(numero)

lista = [5,12,1,6,32,7]
ordenSeleccion(lista)
print(lista)
```

## — Ejecución:

```
PS D:\Ordenamiento\III Semestre\LAB ESTRUCTURAS DE DATOS\ORDENAMIENTO SELECCION> python ordenamientoSeleccion.py
[1, 5, 6, 7, 12, 32]
```

# Ordenamiento por método Shell



## Ordena

Valores de una lista

## Compara

todos los elementos de un arreglo, haciendo saltos entre ellos dependiendo la longitud del arreglo.

## Ventajas

Algoritmo muy simple, uno de los más rápidos, no requiere memoria adicional

## Desventajas

Realiza numerosas comparaciones e intercambios.

# Pasos

Calcular intervalo  $\rightarrow n/2$

Se clasifica cada grupo por separado

Replicar método de inserción

Actualizar intervalo, es decir  $(n/4)$

Concluye cuando se alcanza el tamaño de salto 1

Intervalo =  $n/2$

8	5	3	9	1	4	7
---	---	---	---	---	---	---





## Código

```
1 def ordenamientoShell(lista):
2     contadorSubListas = len(lista)//2
3     while contadorSubListas>0:
4         for posicionInicio in range(contadorSubListas):
5             brechaOrdenamientoInsercion(lista,posicionInicio,contadorSubListas)
6             print("Despues del incremento de tamaño: ", contadorSubListas,"la lista
7                 contadorSubListas=contadorSubListas//2
8
9 def brechaOrdenamientoInsercion(lista,inicio,brecha):
10     for i in range(inicio+brecha,len(lista),brecha):
11         valorActual1 = lista[i]
12         posicion=i
13         while posicion>= brecha and lista[posicion-brecha]>valorActual1:
14             lista[posicion]= lista[posicion-brecha]
15             posicion = posicion-brecha
16         lista[posicion]=valorActual1
17
18 lista=[20,4,36,13,16,19,5,4]
19 print (lista)
20 ordenamientoShell(lista)
21 print("la lista ordenada es: ", lista)
```

## Ejecución:

```
SICION/ordenamientoshell.py
[20, 4, 36, 13, 16, 19, 5, 4]
Despues del incremento de tamaño: 4 la lista es: [16, 4, 5, 4, 20, 19, 36, 13]
Despues del incremento de tamaño: 2 la lista es: [5, 4, 16, 4, 20, 13, 36, 19]
Despues del incremento de tamaño: 1 la lista es: [4, 4, 5, 13, 16, 19, 20, 36]
la lista ordenada es: [4, 4, 5, 13, 16, 19, 20, 36]
PS D:\UNSCH\III semestre\LAB ESTRUCTURA DE DATOS\EXPOSICION> □
```

# Ordenamiento Binario

Este algoritmo, consiste en emplear la búsqueda binaria en lugar de búsqueda lineal. Pues es una modificación del ordenamiento por inserción.

## Ordena

Valores de una lista

## Busca

Una submatriz ordenada y otra sin ordenar, y ubicar la posición correcta de un elemento.

## Ventajas

Eficiencia del algoritmo para pocos datos.

## Desventajas

No funciona bien con una lista grande.

# Pasos para el ordenamiento binario

01

Marcar el primer  
elemento sin ordenar  
A[1] como la clave.

02

Búsqueda binaria para  
encontrar la posición  
de A[1].

03

Desplazar los elementos  
de la posición 1 paso a  
la derecha e insertar  
A[1].

04


Recursividad del  
algoritmo.

# Código

```
def busqueda_binaria(A, c, low, high):
    if high <= low:
        if c > A[low]:
            return low + 1
        else:
            return low
    mid = int((low+high)/2)
    if c == A[mid]:
        return mid
    if c > A[mid]:
        return busqueda_binaria(A,c,mid+1,high)
    else:
        return busqueda_binaria(A,c,low,mid-1)

def ordenamiento_binario(A,n):
    for i in range (1,n):
        j = i-1
        clave = A[i]
        p = busqueda_binaria(A,clave,0,j)
        while j >= p:
            A[j+1] = A[j]
            j = j-1
        A[j+1] = clave
    print(A)
A = [5,3,4,2,1,6]
print(A)
n=6
ordenamiento_binario(A,n)
```

Salida del código:



[5, 3, 4, 2, 1, 6]

[1, 2, 3, 4, 5, 6]

# Ordenamiento Rápido (Quicksort)

Este algoritmo, consiste en dividir el array en dos partes alrededor de un elemento pivote.

Se basa en el principio del algoritmo de dividir y conquistar.

## Ordena

Valores de una lista

## Mueve

Los elementos más pequeños a la izquierda del pivote y los más grandes a la derecha.

## Ventajas

Una alta eficiencia, debido a que es capaz de tratar con una enorme lista de elementos. Debido a que ordena en el lugar, tampoco requiere de almacenamiento adicional.

## Desventajas

Su rendimiento en el peor de los casos es similar a los rendimientos promedio del tipo de ordenamiento de burbuja, inserción o por selección.

# Pasos para el ordenamiento rápido

01

## Partition()

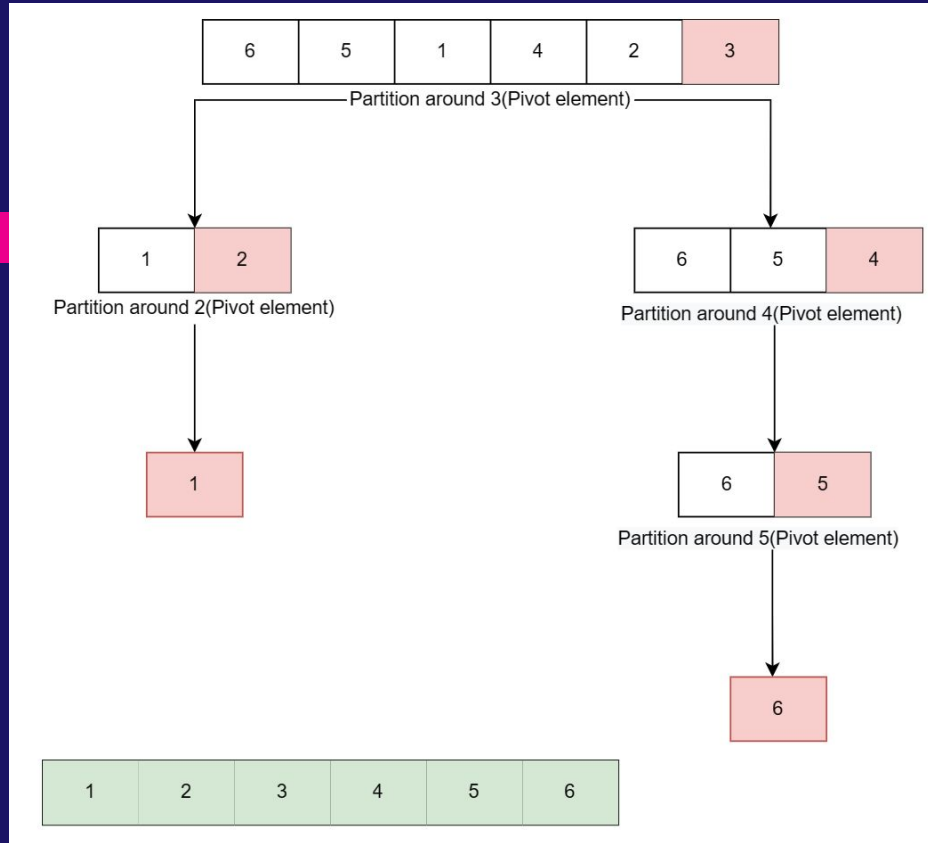
- Selección de un elemento pivote.
- Inicializa el valor de "i" a "beg-1"
- Recorre iterativamente el array
- Evalúa la posición del elemento A[i].

02

## Quicksort()

- Selecciona el índice del pivote.
- Particiona el array.
- Ordenamiento recursivo de los elementos.

# Gráfica:




# Código

```
def partition(A,beg,end):  
    pivot = A[end]  
    i = beg-1  
    for j in range (beg,end):  
        if A[j] <= pivot:  
            i = i + 1  
            A[i], A[j] = A[j], A[i]  
    A[i+1], A[end] = A[end], A[i+1]  
    return i+1
```

```
def quicksort(A,beg,end):  
    if beg < end:  
        pi = partition(A,beg,end)  
        quicksort(A,beg,pi-1)  
        quicksort(A,pi+1, end)
```

```
A = [6,5,1,4,2,3]  
n = 6  
print(A)  
quicksort(A,0,n-1)  
print(A)
```

Salida del código:



[6, 5, 1, 4, 2, 3]

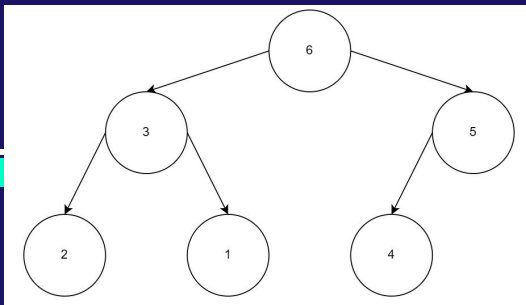
[1, 2, 3, 4, 5, 6]



# Ordenamiento por Montículos (Heapsort)

Este algoritmo, está basado en la comparación y funciona de manera similar al ordenamiento por selección.

Además, el montón es una estructura de datos especial basada en un árbol binario.



## Ordena

Valores de una lista

## Selecciona

El elemento máximo del array usando **max-heap** y lo coloca en su posición final del array.

## Ventajas

La principal ventaja es su simplicidad y buen rendimiento cuando se trabaja con una pequeña lista.

## Desventajas

No recursivo, no estable, con complejidad computacional  $O(n \log n)$

# Pasos para el ordenamiento por montículos

01

## HeapSort()

- Construye un montón máximo.
- A[0] contendrá el elemento máximo.

02

## Heapify()

- Inicializa el índice parent.
- Calcula leftChild y rightChild.
- Compara leftChild con el índice parent.
- Compara rightChild con el índice parent.
- Heapify() recursivo.

# Código

```
def heapify (A,n,i):  
    parent = i  
    leftChild = 2*i+1  
    rightChild = 2*i+2  
    if leftChild < n and A[leftChild] > A[parent]:  
        parent = leftChild  
    if rightChild < n and A[rightChild] > A[parent]:  
        parent = rightChild  
    if parent != i:  
        A[i], A[parent] = A[parent], A[i]  
        heapify(A,n,parent)  
  
def heapSort(A,n):  
    for i in range (int(n/2-1), -1, -1):  
        heapify(A,n,i)  
    for i in range (n-1, 0, -1):  
        A[0], A[i] = A[i], A[0]  
        heapify(A,i,0)  
  
A = [5,3,4,2,1,6]  
print(A)  
n = 6  
heapSort(A,n)  
print(A)
```

Salida del código:

[5, 3, 4, 2, 1, 6]

[1, 2, 3, 4, 5, 6]

# Ordenamiento Radix

Este algoritmo, consiste en evitar las comparaciones insertando elementos en cubos de acuerdo con el radix (base).

## Ordena

El algoritmo por medio del conteo de los dígitos desde el menos significativo al más significativo.

## Inserta

Elementos en cubos de acuerdo con el radix.

## Ventajas

La principal ventaja es ser un ordenamiento no comparativo, lo cuál es muy útil en máquinas paralelas

## Desventajas

Su eficiencia se ve afectada si el número de dígitos en las llaves es demasiado grande.

## Pasos para el ordenamiento radix

01

Encontrar el elemento más grande dentro del array.

02

Ordenar cada dígito empezando por el menos significativo.


# Código

```
import math
class Radix:
    def countingSort(self, A, digit, radix):
        B = [0]*len(A)
        C = [0]*int(radix)
        for i in range(0,len(A)):
            digitAi = (A[i]/radix**digit)%radix
            C[int(digitAi)] = C[int(digitAi)]+1
        for j in range(1,radix):
            C[j] = C[j] + C[j-1]
        for m in range (len(A)-1,-1,-1):
            digitAi = (A[m]/radix**digit)%radix
            C[int(digitAi)] = C[int(digitAi)]-1
            B[C[int(digitAi)]] = A[m]
        return B

    def radixsort(self,A,radix):
        m = max(A)
        output = A
        digits = int(math.floor(math.log(m,radix)+1))
        for i in range(digits):
            output = self.countingSort(output,i,radix)
        return output

a = Radix()
A = [1851,913,1214,312,111,23,41,9]
print(A)
print (a.radixsort(A,10))
```

Salida del código:



```
[1851, 913, 1214, 312, 111, 23, 41, 9]
[9, 23, 41, 111, 312, 913, 1214, 1851]
```

The background is a dark blue gradient with several horizontal bars in white, cyan, and magenta. The word 'THANKS!' is centered in a large, bold, white sans-serif font, set against a magenta rectangular background.

**THANKS!**