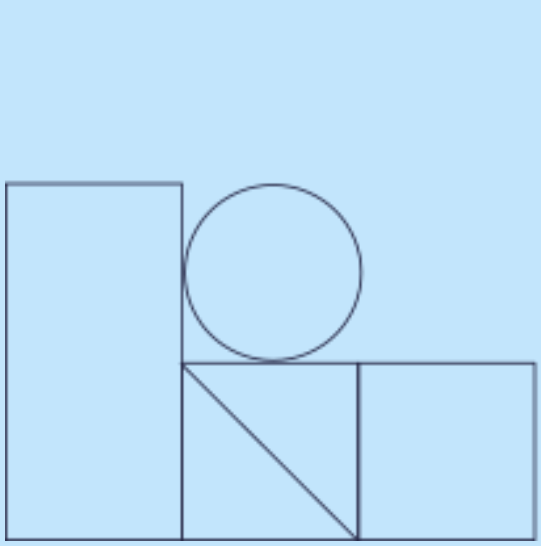
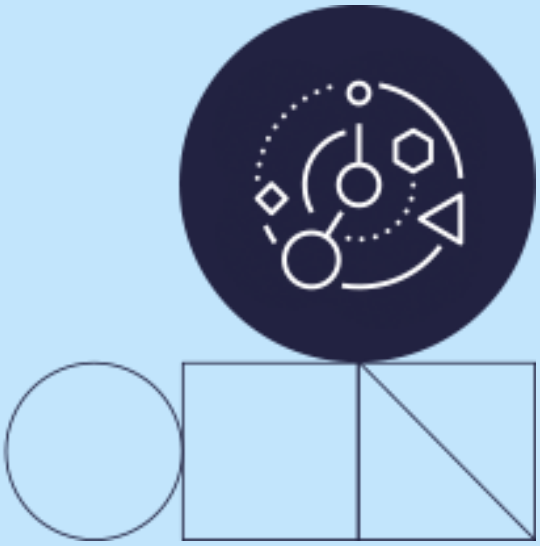
IBM SkillsBuild

IBM

# Programando en Python

## Algoritmos de clasificación en Python





# Índice

# Introducción

La clasificación es una de las funciones más utilizadas en programación. Y si no usamos el algoritmo correcto llevará tiempo extra completar la clasificación.

En este tema, veremos diferentes algoritmos de clasificación.

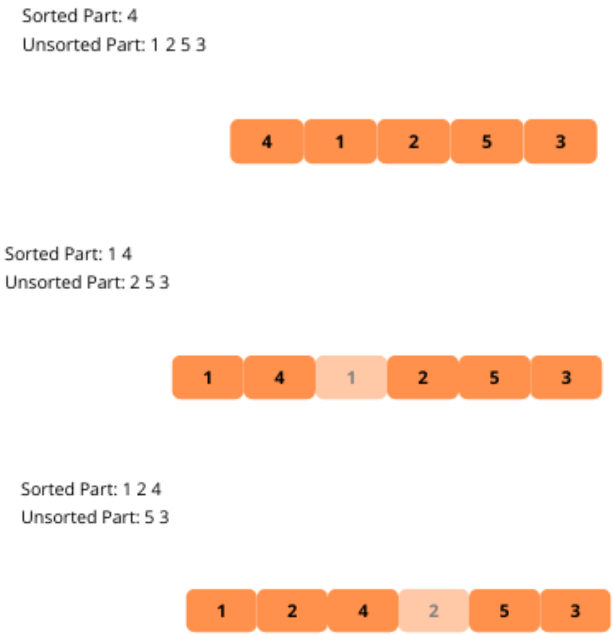
# Insertion Sort

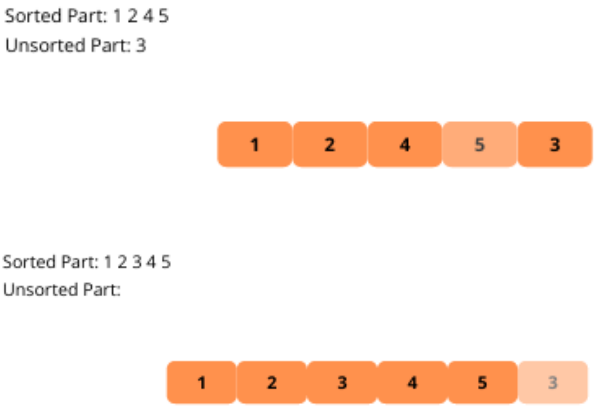
La ordenación por inserción es uno de los algoritmos de ordenación más simples. Es fácil de implementar, pero a la hora de ordenar matrices necesitará más tiempo. No es recomendable para clasificar matrices grandes.

El algoritmo mantiene subpartes ordenadas y no ordenadas en la matriz dada.

La subparte contiene solo el primer elemento al comienzo del proceso de clasificación. Tomaremos un elemento de la matriz no ordenada y los colocaremos en la posición correcta en la sub-matriz ordenada.

Veamos las ilustraciones visuales de tipo de inserción paso a paso con un ejemplo.




Veamos los pasos para implementar el **tipo de inserción**.

- Inicialice la matriz con datos ficticios (enteros).
- Itere sobre la matriz dada desde el segundo elemento.
  - Tome la posición actual y el elemento en dos variables.
  - Escriba un bucle que se repita hasta que aparezca el primer elemento de la matriz o el elemento que sea menor que el elemento actual.
    - Actualiza el elemento actual con el elemento anterior.
    - Disminución de la posición actual.
  - Aquí, el bucle debe llegar al inicio de la matriz o encontrar un elemento más pequeño que el elemento actual. Reemplaza el elemento de posición actual con el elemento actual.

La complejidad temporal del **tipo de inserción** is **O (n ^ 2),** y la complejidad del espacio si **O (1)**.

Hemos ordenado la matriz dada. Ejecutemos el siguiente código.

```
def insertion_sort(arr, n):
    for i in range(1, n):

        ## posición actual y elemento

        current_position = i
        current_element = arr[i]

        ## iterar hasta llega al primer
elemento o
        ## el elemento actual es más
pequeño que el elemento anterior

        while current_position > 0 and
current_element <
         arr[current_position - 1]:
            ## actualizar el elemento
actual con el elemento anterior
            arr[current_position] =
arr[current_position - 1]

            ## moviéndose a la posición
anterior
            current_position -= 1

        ## actualizar el elemento de
posición actual
        arr[current_position] =
current_element

if __name__ == '__main__':
    ## inicialización del array
    arr = [3, 4, 7, 8, 1, 9, 5, 2, 6]
    insertion_sort(arr, 9)

    ## imprimiendo el array

    print(str(arr))
```

# Selection Sort

El orden de selección es similar al orden de inserción con una ligera diferencia. Este algoritmo también divide la matriz en subpartes ordenadas y no ordenadas. Y luego, en cada iteración, tomaremos el elemento mínimo de la subparte sin clasificar y lo colocaremos en la última posición de la subparte ordenada.

Sorted Part:
Unsorted Part: 4 1 2 5 3


Sorted Part: 1
Unsorted Part: 4 2 5 3


Sorted Part: 1 2
Unsorted Part: 4 5 3


Sorted Part: 1 2 3
Unsorted Part: 4 5


Sorted Part: 1 2 3 4
Unsorted Part: 5


Sorted Part: 1 2 3 4 5
Unsorted Part:

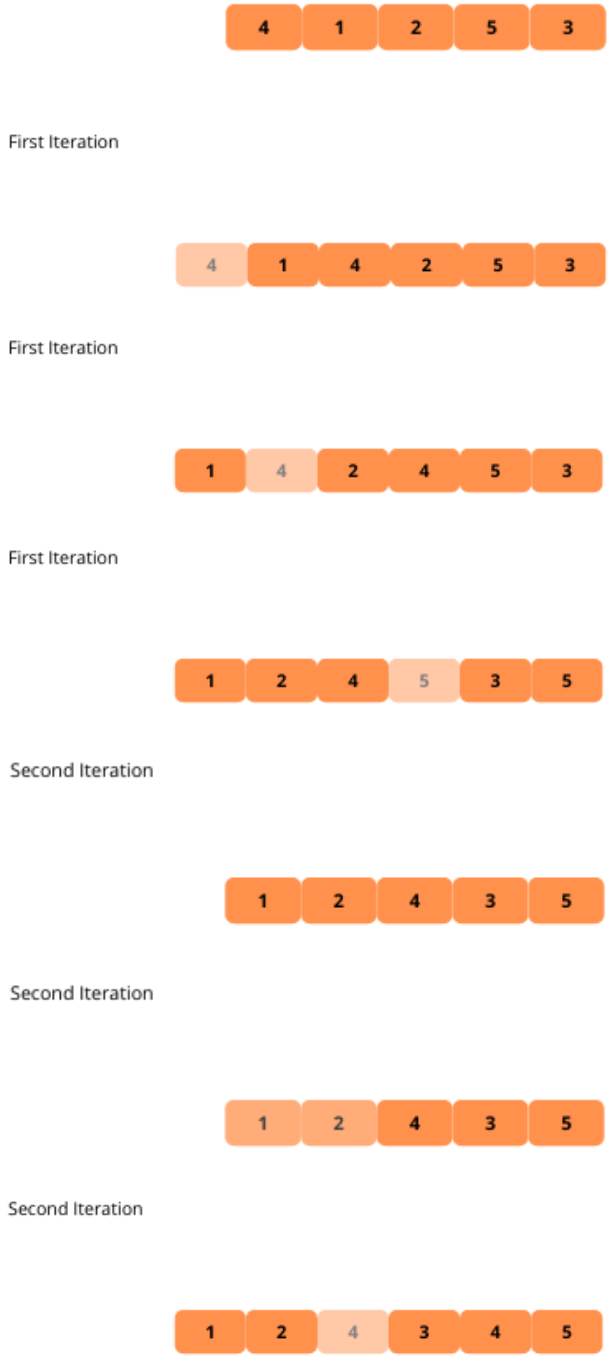

Veamos los pasos para implementar el **tipo de selección**.

- Inicialice la matriz con datos ficticios (enteros).
- Itere sobre la matriz dada.
  - Mantener el índice del elemento mínimo.
  - Escribe un bucle que repita desde el elemento actual hasta el último elemento.
    - Compruebe si el elemento actual es menor que el elemento mínimo o no.
    - Si el elemento actual es menor que el elemento mínimo, reemplace el índice.
  - Tenemos el índice mínimo de elementos con nosotros. Intercambia el elemento actual con el elemento mínimo usando los índices.

La complejidad temporal del **tipo de selección** is **O (n ^ 2),** y la complejidad del espacio si **O (1)**.

Podemos ver el código de implementación del algoritmo a continuación.

```
def selection_sort(arr, n):
    for i in range(n):

        ## para almacenar el índice del
elemento mínimo
        min_element_index = i
        for j in range(i + 1, n):
            ## comprobando y reemplazando
el índice mínimo del elemento
            if arr[j] <
arr[min_element_index]:
                min_element_index = j

        ## intercambiando el elemento
actual con el elemento mínimo
        arr[i], arr[min_element_index] =
arr[min_element_index], arr[i]

if __name__ == '__main__':
    ## inicialización del array

    arr = [3, 4, 7, 8, 1, 9, 5, 2, 6]
    selection_sort(arr, 9)

    ## imprimiendo el array

    print(str(arr))
```




# Bubble Sort

La clasificación de burbujas es un algoritmo simple. Intercambia los elementos adyacentes en cada iteración repetidamente hasta que se ordena la matriz dada.

Itera sobre la matriz y mueve el elemento actual a la siguiente posición hasta que es menor que el siguiente elemento.

Las ilustraciones nos ayudan a comprender **ordenamiento de burbuja** visualmente. Veámoslos.

Veamos los pasos para implementar el **ordenamiento de burbuja**.

- Inicialice la matriz con datos ficticios (enteros).
- Itere sobre la matriz dada.
  - Iterar desde **0** a **ni-1**. El último **i** los elementos ya están ordenados.
    - Compruebe si el elemento actual es mayor que el siguiente elemento o no.
    - Si el elemento actual es mayor que el siguiente, intercambie los dos elementos.

La complejidad temporal del **ordenamiento de burbuja** is **O (n ^ 2),** y la complejidad del espacio si **O (1)**.

Podemos ver el código de implementación del algoritmo de clasificación de burbujas a continuación.

```
def bubble_sort(arr, n):
    for i in range(n):
        ## iterando de 0 a n-i-1 ya que
los últimos i elementos ya están
ordenados
        for j in range(n - i - 1):
            ## comprobando el siguiente
elemento
            if arr[j] > arr[j + 1]:
                ## intercambiando los
elementos adyacentes
                arr[j], arr[j + 1] =
arr[j + 1], arr[j]

if __name__ == '__main__':
    ## inicialización del array

    arr = [3, 4, 7, 8, 1, 9, 5, 2, 6]
    bubble_sort(arr, 9)

    ## imprimiendo el array

    print(str(arr))
```

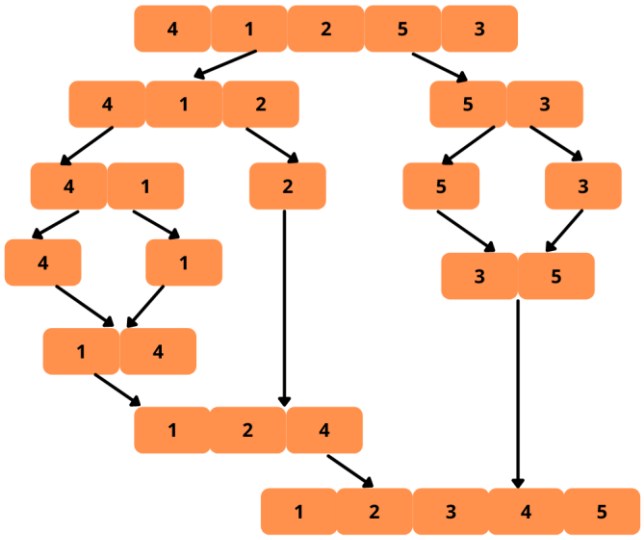
# Merge Sort

El algoritmo de ordenamiento por mezcla (Merge sort) es un algoritmo recursivo para ordenar la matriz dada. Es más eficiente que los algoritmos discutidos anteriormente en términos de complejidad de tiempo. Sigue el enfoque de divide y vencerás.

El algoritmo de clasificación por fusión divide la matriz en dos mitades y las clasifica por separado. Después de ordenar las dos mitades de la matriz, las fusiona en una sola matriz ordenada.

Como es un algoritmo recursivo, divide la matriz hasta que la matriz se convierte en la más simple (matriz con un elemento) para ordenar.

Veamos el ejemplo:




Veamos los pasos para implementar el **tipo de fusión**.

- Inicialice la matriz con datos ficticios (enteros).
- Escribe una función llamada **unir** para fusionar submatrices en una sola matriz ordenada. Acepta la matriz de argumentos, los índices izquierdo, medio y derecho.
    - Obtenga las longitudes de las submatrices izquierda y derecha utilizando los índices dados.
    - Copie los elementos de la matriz en las respectivas matrices izquierda y derecha.
    - Repita las dos submatrices.
        - Compare los dos elementos de las submatrices.
        - Reemplace el elemento de la matriz con el elemento más pequeño de las dos submatrices para ordenar.
    - Compruebe si quedan elementos en ambas submatrices.
    - Agréguelos a la matriz.
- Escribe una función llamada **merge_sort** con matriz de parámetros, índices izquierdo y derecho.
    - Si el índice de la izquierda es mayor o igual que el índice de la derecha, regrese.
    - Encuentre el punto medio de la matriz para dividir la matriz en dos mitades.
    - Llame recursivamente al **merge_sort** utilizando los índices izquierdo, derecho y medio.
    - Después de las llamadas recursivas, combine la matriz con el **unir** función.

La complejidad temporal del **tipo de fusión** es **O (nlogn),** y la complejidad del espacio **O (1)**.

Podemos ver el código de implementación del algoritmo de clasificación por fusión a continuación.

```
def merge(arr, left_index, mid_index,
right_index):
    ## matrices izquierda y derecha
    left_array =
arr[left_index:mid_index+1]
    right_array =
arr[mid_index+1:right_index+1]

    ## obtener las longitudes de matriz
izquierda y derecha
    left_array_length = mid_index -
left_index + 1
    right_array_length = right_index -
mid_index

    ## índices para fusionar dos matrices
    i = j = 0

    ## índice para el reemplazo de
elementos de matriz
    k = left_index

    ## iterando sobre las dos sub-
matrices
    while i < left_array_length and j <
right_array_length:

        ## comparar los elementos de las
matrices izquierda y derecha
        if left_array[i] <
right_array[j]:
            arr[k] = left_array[i]
            i += 1
        else:
            arr[k] = right_array[j]
            j += 1
        k += 1
 ## agregando elementos restantes de las
matrices izquierda y derecha
    while i < left_array_length:
        arr[k] = left_array[i]
        i += 1
        k += 1

    while j < right_array_length:
        j += 1
        k += 1

def merge_sort(arr, left_index,
right_index):
    ## caso base para función recursiva
    if left_index >= right_index:
        return

    ## encontrar el índice medio
    mid_index = (left_index +
right_index) // 2

    ## llamadas recursivas
    merge_sort(arr, left_index,
mid_index)
    merge_sort(arr, mid_index + 1,
right_index)

    ## fusionando las dos sub-matrices
    merge(arr, left_index, mid_index,
right_index)

if __name__ == '__main__':
    ## inicialización de matriz
    arr = [3, 4, 7, 8, 1, 9, 5, 2, 6]
    merge_sort(arr, 0, 8)

    ## imprimiendo la matriz
    print(str(arr))
```