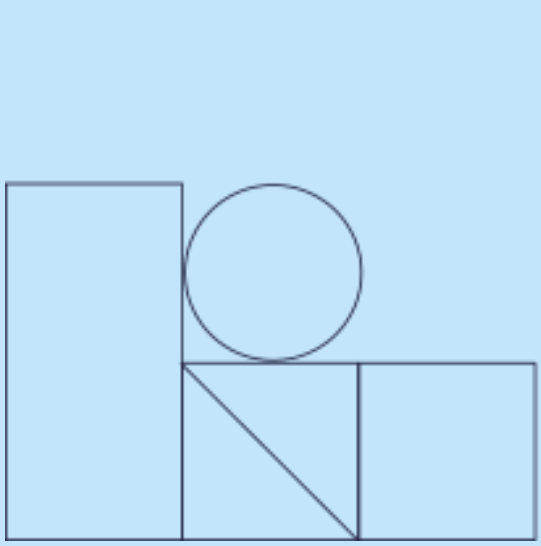
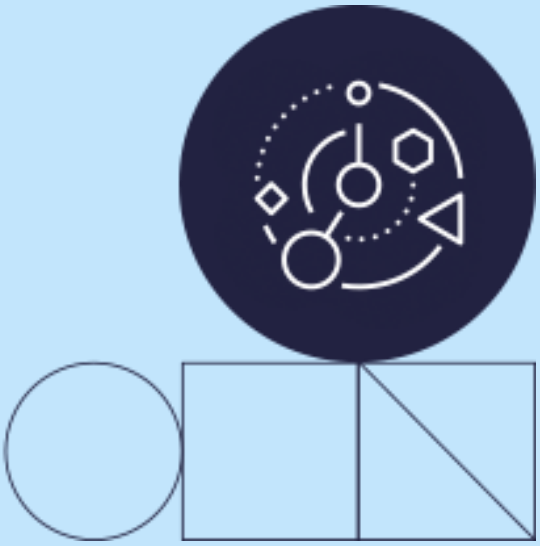
IBM SkillsBuild

IBM

Programação em Python

Algoritmos de classificação em Python





Índice

Introdução

A ordenação é uma das funções mais utilizadas na programação. E se não usarmos o algoritmo correto, levará mais tempo para completar a triagem.

Neste tópico, analisaremos diferentes algoritmos de ordenação.

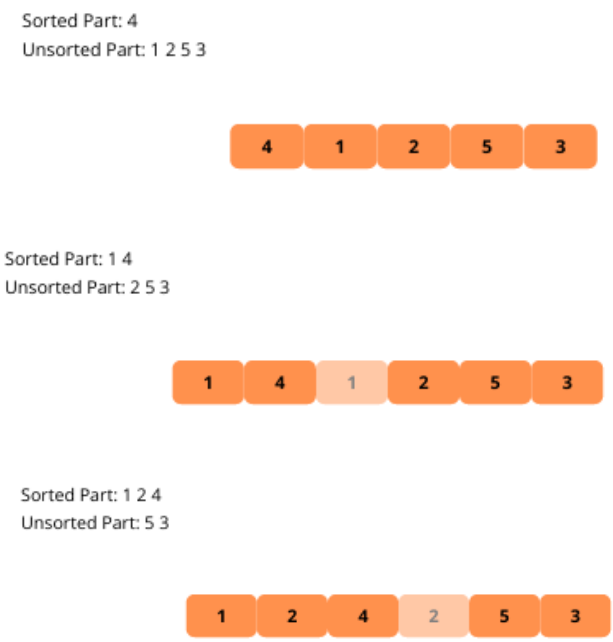
Insertion Sort

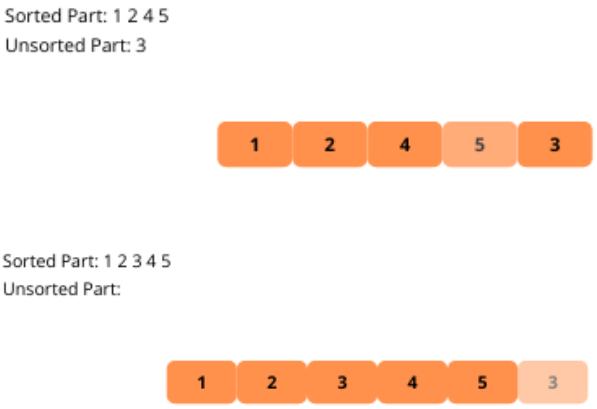
O tipo de inserção é um dos mais simples algoritmos de classificação. É fácil de implementar, mas ao ordenar as matrizes, levará mais tempo. Não é recomendado para a classificação de matrizes grandes.

O tipo de inserção O algoritmo mantém as subpartes ordenadas e não ordenadas na matriz dada.

A subparte contém apenas o primeiro elemento no início do processo de classificação. Pegaremos um elemento da matriz não ordenada e o colocaremos na posição correta na sub-matriz ordenada.

Vejamos as ilustrações visuais do tipo de inserção passo a passo com um exemplo.





Vamos dar uma olhada nos passos para implementar o **tipo de inserção**.

- Inicializar a matriz com dados fictícios (inteiros).
- Iterar sobre a matriz do segundo elemento.
  - Tomar a posição atual e o elemento em duas variáveis.
  - Escreva um laço que se repita até aparecer o primeiro elemento da matriz ou o elemento que é menor que o elemento atual.
    - Atualiza o item atual com o item anterior.
    - Diminuição da posição atual.
  - Aqui, o laço deve chegar ao início da matriz ou encontrar um elemento menor do que o elemento atual. Substituir o elemento de posição atual pelo elemento de posição atual.

A complexidade do tempo do **tipo de inserção** é **O (n ^ 2),** e a complexidade do espaço se **O (1)**.

Classificamos o conjunto dado. Vamos executar o seguinte 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

A ordem de seleção é semelhante à ordem de inserção com uma ligeira diferença. Este algoritmo também divide a matriz em subpartes ordenadas e não ordenadas. E então, em cada iteração, pegamos o elemento mínimo da sub-parte não classificada e o colocamos na última posição da sub-parte classificada.

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:

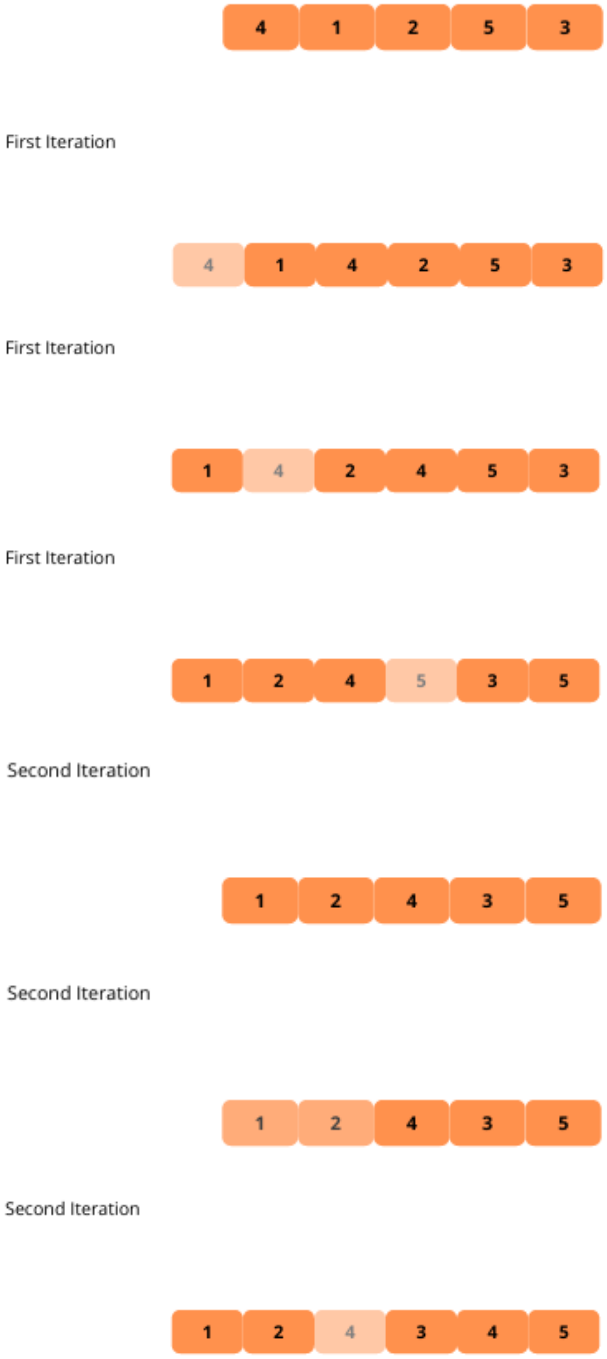
Vamos olhar para as etapas de implementação do **tipo de seleção**.

- Inicializar a matriz com dados fictícios (inteiros).
- Iterar sobre a matriz dada.
  - Manter o índice do elemento mínimo.
  - Escrever um loop que se repete do elemento atual até o último elemento.
    - Verificar se o elemento atual é menor do que o elemento mínimo ou não.
    - Se o elemento atual for menor que o elemento mínimo, substitua o índice.
  - Temos conosco o índice mínimo de elementos. Trocar o elemento atual com o elemento mínimo usando os índices.

A complexidade do tempo do **tipo de seleção** é **O (n ^ 2),** e a complexidade do espaço se **O (1)**.

Podemos ver o código de implementação do algoritmo abaixo.

```
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

A classificação de bolhas é um algoritmo simples. Ele troca os elementos adjacentes a cada iteração repetidamente até que a matriz dada seja classificada.

Ela itera sobre a matriz e move o elemento atual para a próxima posição até que ele seja menor que o próximo elemento.

As ilustrações nos ajudam a entender visualmente **classificação de bolhas**. Vamos olhar para eles.

Vamos olhar para as etapas de implementação do **classificação de bolha**.

- Inicializar a matriz com dados fictícios (inteiros).
- Iterar sobre a matriz dada.
  - Iterar de **0** a **ni-1**. El último **i** os elementos já estão encomendados.
    - Verificar se o item atual é maior do que o próximo item ou não.
    - Se o elemento atual for maior que o elemento seguinte, troque os dois elementos.

A complexidade do tempo do **classificação de bolha** é **O (n ^ 2),** e a complexidade do espaço se **O (1)**.

Podemos ver abaixo o código de implementação do algoritmo de classificação de bolhas.

```
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))
```

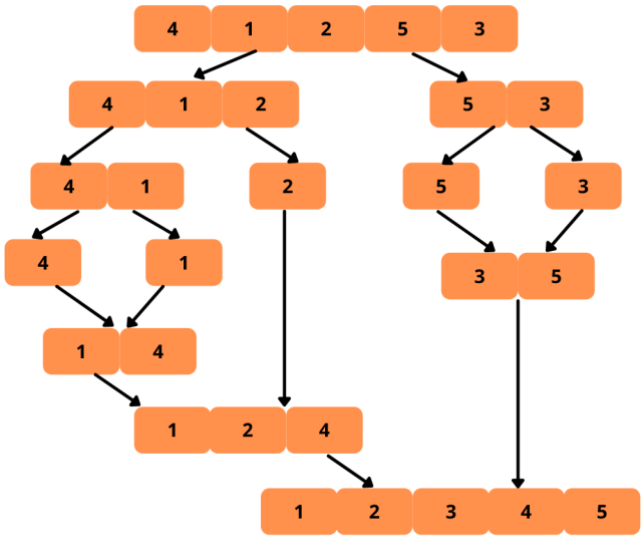
Merge Sort

O algoritmo de classificação da mistura (Merge sort) é um algoritmo recursivo para ordenar a matriz dada. É mais eficiente do que os algoritmos discutidos acima em termos de complexidade de tempo. Ela segue a abordagem de dividir e conquistar.

O algoritmo de separação da fusão divide a matriz em duas metades e as classifica separadamente. Depois de classificar as duas metades da matriz, ela as funde em uma única matriz classificada.

Por ser um algoritmo recursivo, ele divide a matriz até que a matriz se torne a mais simples (matriz de um elemento) de classificar.

Vejamos o exemplo:



Vamos olhar para as etapas de implementação do **tipo de fusão**.

- Inicializar a matriz com dados fictícios (inteiros).
- Escreva uma função chamada **unir** para fundir os subarrays em uma única matriz ordenada. Aceita a matriz de argumentos, índices esquerdo, médio e direito.
  - Obter os comprimentos das submatrices esquerda e direita usando os índices dados.
  - Copiar os elementos da matriz para as respectivas matrizes esquerda e direita.
  - Repita os dois sub-matrices.
    - Compare os dois elementos dos sub-matrizes.
    - Substituir o elemento da matriz pelo elemento menor dos dois sub-matrizes a serem encomendados.
  - Verifique se ainda há elementos em ambos os sub-matrizes.
  - Adicioná-los à matriz.
- Escreva uma função chamada **merge_sort** com matriz de parâmetros, índices esquerdo e direito.
  - Se o índice da esquerda for maior ou igual ao índice da direita, retornar para.
  - Encontrar o ponto médio da matriz para dividir a matriz em duas metades.
  - Recursivamente chamar o **merge_sort** usando os índices da esquerda, direita e do meio.
  - Após as chamadas recursivas, combine a matriz com o **unir** função.

A complexidade do tempo do **tipo de fusão** é **O (nlogn),** e a complexidade do espaço **O (1)**.

Podemos ver abaixo o código de implementação do algoritmo de separação da fusão.

```
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))
```