



Matplotlib

Visualización de datos

Decorative footer elements consisting of several horizontal bars in white, red, and cyan at the bottom of the slide.

1
2
3
4
5
6
7
8
9
10
11
12
13
14

01 {

[Matplotlib]

- Matplotlib
- Instalación

}

Matplotlib {

Matplotlib es una librería de visualización de datos de Python que **permite crear gráficos y visualizaciones de datos de alta calidad de manera sencilla.**

Matplotlib es una de las librerías de visualización de datos más populares y utilizadas en Python, y es ampliamente utilizada en la comunidad científica y de análisis de datos.

Matplotlib también es **altamente integrable con otras librerías de análisis de datos de Python**, como **NumPy** y **Pandas**, lo que lo hace ideal para la visualización de datos científicos y de análisis de datos.

}

Matplotlib {

Matplotlib se puede utilizar para crear una amplia variedad de gráficos, incluyendo **gráficos de línea, gráficos de barras, gráficos de dispersión, gráficos de contorno, gráficos de superficie, gráficos de histograma** y muchos más.

Además, Matplotlib **ofrece una gran cantidad de opciones de personalización** para ajustar el aspecto de los gráficos, como colores, etiquetas y títulos.

}

Matplotlib {

Permite crear y personalizar los tipos de gráficos más comunes, entre ellos:

- Diagramas de barras
- Diagramas de dispersión o puntos
- Histograma
- Diagramas de líneas
- Diagramas de sectores
- Diagramas de áreas
- Diagramas de caja y bigotes
- Diagramas de contorno
- Diagramas de violín
- Mapas de color

}

Instalación {

Para utilizar Matplotlib, primero se debe instalar en tu computadora. Matplotlib se puede instalar utilizando la **gestión de paquetes de Python (pip)**:

```
pip install matplotlib
```

}



Ordenamiento

Algoritmos de ordenamiento

01 {

[Ordenamientos]

- Ordenamiento Burbuja
- Ordenamiento por Inserción
- Ordenamiento por Selección
- Ordenamiento por Mezcla

}

Ordenamiento Burbuja {

El Algoritmo de Ordenamiento por Burbuja (Bubble Sort) es un **algoritmo de ordenamiento básico** que **funciona comparando elementos adyacentes en una lista y cambiando los elementos si están en el orden incorrecto.**

El ordenamiento burbuja se llama así porque el algoritmo funciona comparando elementos adyacentes en una lista y cambiando los elementos si están en el orden incorrecto. El proceso se repite varias veces, hasta que **los elementos más grandes "burbujean" hasta el final de la lista** y los elementos más pequeños quedan al inicio de la lista.

}

Ordenamiento Burbuja {

El algoritmo comienza en el primer elemento de la lista y compara el elemento con el siguiente elemento.

Si el primer elemento es mayor que el segundo elemento, los elementos se **intercambian**.

El algoritmo luego continúa al siguiente elemento y **repite el proceso**.

El algoritmo continúa hasta llegar al último elemento de la lista. Una vez que el algoritmo ha llegado al último elemento, la lista estará ordenada.

}

Ordenamiento Burbuja {

1
2
3
4
5
6
7
8
9
10
11
12
13
14

}

6 5 3 1 8 7 2 4

Ordenamiento Burbuja {

8 4 2 6 9

}

Ordenamiento Burbuja {



```
def bubble_sort(lista):  
    tamaño = len(lista)  
    for i in range(1, tamaño):  
        for j in range(0, tamaño - i):  
            if lista[j] > lista[j + 1]:  
                lista[j], lista[j + 1] = lista[j + 1], lista[j]  
    return lista
```

}

Ordenamiento Burbuja {

La **complejidad algorítmica de bubble sort es $O(n^2)$** , lo que significa que el tiempo de ejecución del algoritmo es proporcional al cuadrado del tamaño de la entrada.

Esto se debe a que el algoritmo debe comparar cada elemento de la lista con cada otro elemento. Para una **lista de tamaño n** , esto significa que el algoritmo debe realizar **n^2** comparaciones.

}

Ordenamiento Burbuja {

El ordenamiento burbuja es un **algoritmo ineficiente y no se recomienda su uso para ordenar listas grandes.**

Sin embargo, es un algoritmo simple y fácil de entender, y puede ser útil para aprender sobre los algoritmos de ordenamiento.

}

Ordenamiento por Inserción {

El algoritmo de **Ordenamiento de Inserción (Insertion Sort)** actúa recorriendo la lista a ordenar, tomando el elemento actual e insertándolo donde debería comparándolo con los elementos que ya ha recorrido.

El algoritmo de inserción es un algoritmo simple y fácil de entender. También es un algoritmo eficiente y puede ser utilizado para ordenar listas grandes.

}

Ordenamiento por Inserción {

Este algoritmo separa la lista en dos partes, **ordenada** y **no ordenada**.

Suponemos que el primer elemento está ordenado, luego pasamos al siguiente elemento que lo vamos a llamar **j**, comparamos **j** con el primero, si es mayor, se queda como está pero si es más pequeño, copiamos el primer elemento en la segunda posición e insertamos **j** como primero.

}

Ordenamiento por Inserción {

6 5 3 1 8 7 2 4

}

Ordenamiento por Inserción {

Insertion Sort

5	1	7	8	6	7	5	3	0	9
---	---	---	---	---	---	---	---	---	---

}

Ordenamiento por Inserción {



```
def insertion_sort(lista):  
    tamaño = len(lista)  
    for i in range(1, tamaño):  
        elemento = lista[i]  
        j = i - 1  
        while j >= 0 and lista[j] > elemento:  
            lista[j + 1] = lista[j]  
            j -= 1  
        lista[j + 1] = elemento  
  
    return lista
```

Ordenamiento por Inserción {

Cuantos más elementos tengamos ordenados, **menos elementos tendremos que examinar.**

En el peor caso con la lista ordenada con el criterio contrario se obtiene una complejidad temporal cuadrática, del orden de **$O(n^2/2)$** con una **lista de n elementos.**

}

Ordenamiento por Selección {

El algoritmo de **Ordenamiento por Selección (Selection Sort)** consiste en **buscar el menor entre todos los elementos no ordenados y colocarlo al principio**, luego se debe repetir lo mismo con los restantes (no se tienen en cuenta los ya ordenados).

}

Ordenamiento por Selección {

Este algoritmo separa la lista en dos partes, **ordenada** y **no ordenada**. Continuamente **elimina el elemento más pequeño de la parte sin ordenar y lo agrega a la parte ordenada**.

Lo que realmente realiza este algoritmo es tratar la parte **izquierda de la lista como la parte ordenada** buscando en toda la lista el elemento más pequeño y poniéndolo el primero.

}

Ordenamiento por Selección {

Después, **sabiendo que ya tenemos el elemento más pequeño de primero**, buscamos en toda la lista el elemento más pequeño de los restantes sin ordenar y lo intercambiamos con el siguiente ordenado y así hasta acabar con la lista.

En el peor caso, su complejidad algorítmica sería **$O(n^2)$** .

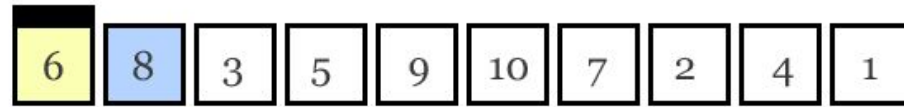
}

Ordenamiento por Selección {

8	5	2	6	9	3	1	4	0	7
---	---	---	---	---	---	---	---	---	---

}

Ordenamiento por Selección {



Yellow is smallest number found
Blue is current item
Green is sorted list


Ordenamiento por Selección {

1
2
3
4
5
6
7
8
9
10
11
12
13
14

}

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Ordenamiento por Selección {



```
def selection_sort(lista):
    tamaño = len(lista)
    for i in range(tamaño - 1):
        posicion_minimo = i
        for j in range(i + 1, tamaño):
            if lista[j] < lista[posicion_minimo]:
                posicion_minimo = j
        lista[i], lista[posicion_minimo] = lista[posicion_minimo], lista[i]

    return lista
```

}

Ordenamiento por Mezcla {

El algoritmo de **Ordenamiento por Mezcla (Merge Sort)** tiene un funcionamiento muy particular, primero debemos saber que **si la longitud de la lista es 0 o 1, ya está ordenada.**

De no ser caso, el algoritmo deberá **dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.**

Luego, ordenará cada sublista recursivamente aplicando el **ordenamiento por mezcla** y por último mezcla las dos sublistas en una sola lista ordenada.

}

Ordenamiento por Mezcla {

Este algoritmos es **recursivo** y ordena los elementos con una metodología de **divide y conquistaras**.

Este algoritmo comienza **dividiendo la lista en dos**, luego esas dos mitades en cuatro y así sucesivamente **hasta que tengamos listas de un elemento de longitud**.

Luego, estos elementos se volverán a unir en orden. Primero, fusionaremos los elementos individuales en pares de nuevo ordenándolos entre sí, luego **seguiremos ordenándolos en grupos hasta que tengamos una sola lista ordenada**.

En el peor caso, su complejidad es **$O(n\log(n))$** .

}

Ordenamiento por Mezcla {

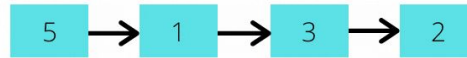
6 5 3 1 8 7 2 4

}

Ordenamiento por Mezcla {

1
2
3
4
5
6
7
8
9
10
11
12
13
14

}



Ordenamiento por Mezcla {



```
def merge_sort(lista):  
    # Si la lista tiene 1 o ningun elemento, se retorna  
    if (len(lista) <= 1):  
        return lista  
    # Posicion en la mitad de la lista  
    posicion_medio = len(lista) // 2  
    # Ordenamos y fusionamos cada mitad  
    lista_izquierda = merge_sort(lista[ : posicion_medio])  
    lista_derecha = merge_sort(lista[posicion_medio : ])  
    # Fusionamos las listas ordenadas en un nueva ordenada  
    return merge(lista_izquierda, lista_derecha)
```

}

ordenamientos.py

```
def merge(lista_izquierda, lista_derecha):
    # Creamos una lista vacia
    lista_ordenada = []
    # Inicializamos en el primer elemento de cada lista
    posicion_izquierda = 0
    posicion_derecha = 0
    # Guardamos el tamaño de cada lista
    if(lista_izquierda == None):
        lista_izquierda = []
    if(lista_derecha == None):
        lista_derecha = []
    tamaño_izquierda = len(lista_izquierda)
    tamaño_derecha = len(lista_derecha)
    for i in range(tamaño_izquierda + tamaño_derecha):
        if(posicion_izquierda < tamaño_izquierda and posicion_derecha < tamaño_derecha):
            # Comprobamos el menor de ambas listas, si es el de la izquierda se añade a la lista ordenada
            if (lista_izquierda[posicion_izquierda] <= lista_derecha[posicion_derecha]):
                lista_ordenada.append(lista_izquierda[posicion_izquierda])
                # Aumentamos la posicion izquierda
                posicion_izquierda += 1
            # Si el menor es el de la derecha, se añade a la lista ordenada
            else:
                lista_ordenada.append(lista_derecha[posicion_derecha])
                # Aumentamos la posicion derecha
                posicion_derecha += 1
        # Si llegamos al final de la lista izquierda, entonces agregamos los elementos de la lista derecha
        elif(posicion_izquierda == tamaño_izquierda):
            lista_ordenada.append(lista_derecha[posicion_derecha])
            # Aumentamos la posicion derecha
            posicion_derecha += 1
        # Si llegamos al final de la lista derecha, entonces agregamos los elementos de la lista izquierda
        elif(posicion_derecha == tamaño_derecha):
            lista_ordenada.append(lista_izquierda[posicion_izquierda])
            # Aumentamos la posicion izquierda
            posicion_izquierda += 1
    return lista_ordenada
```

Algoritmos de Ordenamiento

Merge Sort

