Contenido

```
#Tuplas
·Son similares a las listas, pero la principal diferencia es que
las tuplas son inmutables lo que significa que no se pueden modificar una
vez que se han creado se definen utilizando paréntesis ()
mi_tupla = (1, 2, 3, 'cuatro', 5.0)
-Puedes acceder a los elementos de una tupla mediante índices
print(mi_tupla[0])  # Imprime 1
print(mi_tupla[3])  # Imprime 'cuatro'
```

```
#Listas
·Son mutables, lo que significa que puedes modificar, agregar o eliminar
mi_lista = [1, 2, 3, 'cuatro', 5.0]
```

```
#Diccionarios
·Almacenan datos en pares clave-valor y son útiles cuando necesitas acceder
a valores mediante una clave en lugar de un índice.
mi_diccionario = {'clave1': 'valor1', 'clave2': 'valor2', 'clave3':
'valor3'}
```

```
#Conjuntos
·Son colecciones no ordenadas de elementos únicos. Se utilizan cuando solo
te importa la presencia o ausencia de un elemento, no su posición o
frecuencia.
mi_conjunto = {1, 2, 3, 4, 5}
```

```
#Funciones y métodos comunes que puedes usar con listas
```

```
-append(elemento)
·Agrega un elemento al final de la lista.
mi_lista = [1, 2, 3]
mi_lista.append(4)
print(mi_lista)  # Imprime [1, 2, 3, 4]
```

```
-extend(iterable)
·Extiende la lista agregando elementos de un iterable al final.
mi_lista = [1, 2, 3]
mi_lista.extend([4, 5, 6])
print(mi_lista)  # Imprime [1, 2, 3, 4, 5, 6]
```

```
-insert(posición, elemento)
·Inserta un elemento en una posición específica de la lista.
mi_lista = [1, 2, 3]
mi_lista.insert(1, 1.5)
print(mi_lista)  # Imprime [1, 1.5, 2, 3]
```

```
-pop([índice])
·Elimina y devuelve el elemento en la posición dada (o el último elemento si no se proporciona un índice).
mi_lista = [1, 2, 3]
elemento_eliminado = mi_lista.pop(1)
print(mi_lista)          # Imprime [1, 3]
print(elemento_eliminado) # Imprime 2
```

```
-index(elemento)
·Devuelve el índice de la primera ocurrencia del elemento.
mi_lista = [1, 2, 3, 2]
indice = mi_lista.index(2)
print(indice)  # Imprime 1
```

```
#Algoritmos de Búsqueda:
```

```
-Búsqueda Lineal:
·Recorre la lista elemento por elemento hasta encontrar el valor buscado.

def busqueda_lineal(lista, elemento):
    for i in range(len(lista)):
        if lista[i] == elemento:
            return i  # Devuelve la posición del elemento si se encuentra
    return -1  # Devuelve -1 si el elemento no está en la lista

# Ejemplo de uso
lista_ejemplo = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
elemento_buscado = 7
resultado = busqueda_lineal(lista_ejemplo, elemento_buscado)

print(f"Elemento {elemento_buscado} encontrado en la posición {resultado}")
```

```
-Búsqueda Binaria:
·Funciona en listas ordenadas. Divide la lista en dos y compara el elemento buscado con el valor medio. Luego, se reduce la búsqueda a la mitad correcta.
```

```python
def busqueda_binaria(lista, elemento):
    izquierda, derecha = 0, len(lista) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == elemento:
            return medio  # Devuelve la posición del elemento si se
encuentra
        elif lista[medio] < elemento:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    return -1  # Devuelve -1 si el elemento no está en la lista

# Ejemplo de uso
lista_ordenada = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
elemento_buscado = 7
resultado = busqueda_binaria(lista_ordenada, elemento_buscado)
print(f"Elemento {elemento_buscado} encontrado en la posición {resultado}")
```

```
-Búsqueda Interpolada:
·Similar a la búsqueda binaria, pero utiliza una fórmula matemática para
estimar la posición del elemento buscado.

def busqueda_interpolada(lista, elemento):
    izquierda, derecha = 0, len(lista) - 1

    while izquierda <= derecha and lista[izquierda] <= elemento <=
lista[derecha]:
        # Fórmula de interpolación
        posicion = izquierda + int(((elemento - lista[izquierda]) /
(lista[derecha] - lista[izquierda])) * (derecha - izquierda))

        if lista[posicion] == elemento:
            return posicion  # Devuelve la posición del elemento si se
encuentra
        elif lista[posicion] < elemento:
            izquierda = posicion + 1
        else:
            derecha = posicion - 1

    return -1  # Devuelve -1 si el elemento no está en la lista
```

```python
# Ejemplo de uso
lista_ordenada = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
elemento_buscado = 7
resultado = busqueda_interpolada(lista_ordenada, elemento_buscado)
print(f"Elemento {elemento_buscado} encontrado en la posición {resultado}")
```

```
-Búsqueda Exponencial:
·Trabaja en listas ordenadas y duplica repetidamente el índice de búsqueda
hasta encontrar el valor o superar el tamaño de la lista.
```

```python
def busqueda_exponencial(lista, elemento):
    if lista[0] == elemento:
        return 0

    i = 1
    n = len(lista)

    while i < n and lista[i] <= elemento:
        i *= 2

    return busqueda_binaria(lista[:min(i, n)], elemento)

# Ejemplo de uso
lista_ordenada = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
elemento_buscado = 7
resultado = busqueda_exponencial(lista_ordenada, elemento_buscado)
print(f"Elemento {elemento_buscado} encontrado en la posición {resultado}")
```

```
#Algoritmos de Ordenamiento:
```

```
-Bubble Sort (Ordenamiento de Burbuja):
·Compara y swap sucesivamente los elementos adyacentes hasta que la lista
esté ordenada.
```

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
```

```
bubble_sort(arr)
print("Lista ordenada por Bubble Sort:", arr)
```

```
-Insertion Sort (Ordenamiento por Inserción):
·Construye una secuencia ordenada de elementos uno a uno tomando elementos
de la lista y colocándolos en la posición correcta.

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
insertion_sort(arr)
print("Lista ordenada por Insertion Sort:", arr)
```

```
-Selection Sort (Ordenamiento por Selección):
·Encuentra el elemento mínimo de la lista y lo intercambia con el primer
elemento. Luego, encuentra el mínimo de la lista restante y lo intercambia
con el segundo elemento, y así sucesivamente.

def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
selection_sort(arr)
print("Lista ordenada por Selection Sort:", arr)
```

```
-Merge Sort (Ordenamiento por Mezcla):
·Divide la lista en mitades, ordena cada mitad y luego fusiona las dos
mitades ordenadas para obtener una lista ordenada.

def merge_sort(arr):
    if len(arr) > 1:
```

```
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
merge_sort(arr)
print("Lista ordenada por Merge Sort:", arr)
```

```
-Quick Sort (Ordenamiento Rápido):
·Selecciona un elemento llamado "pivote" y organiza los elementos alrededor
del pivote, dividiendo la lista en dos subconjuntos que se ordenan
recursivamente.

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        lesser = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
```

```
        return quick_sort(lesser) + [pivot] + quick_sort(greater)
# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
arr = quick_sort(arr)
print("Lista ordenada por Quick Sort:", arr)
```

-Heap Sort (Ordenamiento por Montículos):
·Construye un montículo (árbol binario completo) y luego extrae sucesivamente el elemento máximo (o mínimo), reconstruyendo el montículo.

```
def heapify(arr, n, i):
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    if left_child < n and arr[i] < arr[left_child]:
        largest = left_child

    if right_child < n and arr[largest] < arr[right_child]:
        largest = right_child

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
heap_sort(arr)
print("Lista ordenada por Heap Sort:", arr)
```

-Radix Sort (Ordenamiento por Radix):
·Ordena los elementos procesando dígitos individuales de los números en lugar de comparar números completos.

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

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

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

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

    i = 0
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_num = max(arr)
    exp = 1

    while max_num // exp > 0:
        counting_sort(arr, exp)
        exp *= 10
# Ejemplo de uso
arr = [170, 45, 75, 90, 802, 24, 2, 66]
radix_sort(arr)
print("Lista ordenada por Radix Sort:", arr)
```

```
#Algoritmos de Insercion
```

```
-Insertion Sort (Ordenamiento por Inserción):
·Este es el algoritmo más básico y común de ordenamiento por inserción.
Funciona construyendo una secuencia ordenada de elementos tomando uno a uno
los elementos de la lista y colocándolos en la posición correcta.
```

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

```
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
insertion_sort(arr)
print("Lista ordenada por Insertion Sort:", arr)
```

-Shell Sort (Ordenamiento de Shell):
·Es una mejora del algoritmo de inserción que compara elementos distantes entre sí, en lugar de adyacentes, y luego reduce gradualmente la brecha entre elementos a medida que la lista se ordena.

```
def shell_sort(arr):
    n = len(arr)
    intervalo = n // 2

    while intervalo > 0:
        for i in range(intervalo, n):
            temp = arr[i]
            j = i
            while j >= intervalo and arr[j - intervalo] > temp:
                arr[j] = arr[j - intervalo]
                j -= intervalo
            arr[j] = temp
        intervalo //= 2

# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
shell_sort(arr)
print("Lista ordenada por Shell Sort:", arr)
```

-Binary Insertion Sort (Ordenamiento por Inserción Binaria):
·Similar al algoritmo de inserción estándar, pero utiliza búsqueda binaria para encontrar la posición correcta para insertar un elemento en lugar de comparaciones lineales.

```
def binary_insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
```

```
        izquierda, derecha = 0, i - 1

        while izquierda <= derecha:
            medio = (izquierda + derecha) // 2
            if key < arr[medio]:
                derecha = medio - 1
            else:
                izquierda = medio + 1

        for j in range(i - 1, izquierda - 1, -1):
            arr[j + 1] = arr[j]

        arr[izquierda] = key

# Ejemplo de uso
arr = [64, 34, 25, 12, 22, 11, 90]
binary_insertion_sort(arr)
print("Lista ordenada por Binary Insertion Sort:", arr)
```