Ordenamiento por inserción (Insertion Sort) en Python

¿Qué es?

El **ordenamiento por inserción** es un algoritmo que construye la lista ordenada elemento a elemento. Toma cada elemento y lo inserta en su posición correcta dentro de la porción ya ordenada a la izquierda.

- Es **estable** (mantiene el orden relativo de elementos iguales).
- Es **in-place** (no usa arreglo extra significativo).
- Complejidad: **O(n²)** en el peor caso, **O(n)** si la lista ya está casi ordenada.

Código (ejemplo en Python)

```
def insertion_sort(arr):
    # recorre desde el segundo elemento hasta el final
    for i in range(1, len(arr)):
        key = arr[i]           # 1) elemento a insertar
        j = i - 1              # 2) índice del último elemento de la sublista
ordenada
        # desplaza elementos mayores que key una posición a la derecha
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        # coloca key en su lugar correcto
        arr[j + 1] = key

# ejemplo de uso
lista = [5, 2, 4, 6, 1, 3]
insertion_sort(lista)
print(lista)  # salida: [1, 2, 3, 4, 5, 6]
```

Explicación línea por línea

Voy a numerar las líneas conceptualmente (no tienes que poner números en tu archivo):

1. `def insertion_sort(arr):`
   - Declara la función `insertion_sort` que recibe una lista `arr`. Aquí empieza la definición.
2. `# recorre desde el segundo elemento hasta el final`
   - Comentario que explica lo que hace la siguiente línea.
3. `for i in range(1, len(arr)):`
   - Bucle `for` que itera `i` desde 1 hasta `len(arr)-1`.
   - Se empieza en 1 porque el subarray `arr[0:1]` (el primer elemento) ya se considera ordenado.
4. `key = arr[i] # 1) elemento a insertar`
   - Guarda en `key` el elemento en la posición `i` que queremos insertar en la parte ordenada izquierda.
5. `j = i - 1 # 2) índice del último elemento de la sublista ordenada`
   - Inicializa `j` al índice inmediatamente a la izquierda de `i`. `j` se usará para recorrer la sublista ordenada hacia la izquierda.
6. `# desplaza elementos mayores que key una posición a la derecha`
   - Comentario que anticipa el `while`.

7. `while j >= 0 and arr[j] > key:`
    - Mientras no nos salgamos de la lista (`j >= 0`) **y** el elemento `arr[j]` sea mayor que `key`, repetimos el bloque: esto significa que `arr[j]` debe moverse a la derecha porque `key` debe ir antes que él.
8. `arr[j + 1] = arr[j]`
    - Desplaza el elemento `arr[j]` una posición a la derecha (a `j+1`) dejando "hueco" donde luego irá `key`.
9. `j -= 1`
    - Decrementa `j` para comparar `key` con el siguiente elemento a la izquierda en la siguiente iteración del `while`.
10. `# coloca key en su lugar correcto`
    - Comentario.
11. `arr[j + 1] = key`
    - Una vez que el `while` termina, `j` apunta a la posición inmediatamente anterior al lugar correcto para `key` (o a `-1` si `key` es el menor). Colocamos `key` en `j+1`, su posición ordenada.
12. (fuera de la función) `lista = [5, 2, 4, 6, 1, 3]`
    - Creamos una lista de ejemplo.
13. `insertion_sort(lista)`
    - Llamamos a la función para ordenar `lista` in-place.
14. `print(lista) # salida: [1, 2, 3, 4, 5, 6]`
    - Imprime la lista ya ordenada.

Trace paso a paso con `lista = [5, 2, 4, 6, 1, 3]`

Estado inicial: `[5, 2, 4, 6, 1, 3]`

- i = 1 (key = 2), j = 0
    - arr[0] = 5 > 2 → desplazar: `[5, 5, 4, 6, 1, 3]`, j → -1
    - insertar key en j+1 = 0 → `[2, 5, 4, 6, 1, 3]`
- i = 2 (key = 4), j = 1
    - arr[1] = 5 > 4 → desplazar: `[2, 5, 5, 6, 1, 3]`, j → 0
    - arr[0] = 2 <= 4 → salir while
    - insertar en j+1 = 1 → `[2, 4, 5, 6, 1, 3]`
- i = 3 (key = 6), j = 2
    - arr[2] = 5 <= 6 → no desplaza
    - insertar en j+1 = 3 → `[2, 4, 5, 6, 1, 3]` (sin cambios)
- i = 4 (key = 1), j = 3
    - arr[3] = 6 > 1 → `[2, 4, 5, 6, 6, 3]`, j → 2
    - arr[2] = 5 > 1 → `[2, 4, 5, 5, 6, 3]`, j → 1
    - arr[1] = 4 > 1 → `[2, 4, 4, 5, 6, 3]`, j → 0
    - arr[0] = 2 > 1 → `[2, 2, 4, 5, 6, 3]`, j → -1
    - insertar en j+1 = 0 → `[1, 2, 4, 5, 6, 3]`

- i = 5 (key = 3), j = 4
    - arr[4] = 6 > 3 → `[1, 2, 4, 5, 6, 6]`, j → 3
    - arr[3] = 5 > 3 → `[1, 2, 4, 5, 5, 6]`, j → 2
    - arr[2] = 4 > 3 → `[1, 2, 4, 4, 5, 6]`, j → 1
    - arr[1] = 2 <= 3 → salir while
    - insertar en j+1 = 2 → `[1, 2, 3, 4, 5, 6]`

Resultado final: `[1, 2, 3, 4, 5, 6]`.

Consejos prácticos

- Es muy eficiente para listas pequeñas o casi ordenadas.
- Si la lista es grande y está desordenada, algoritmos como **mergesort** o **quicksort** suelen ser mejores (O(n log n)).
- Si quieres mantener la lista original intacta, primero copia: `a = arr.copy()` y ordena `a`.